USENIX

Proceedings of the 5th USENIX Conference on File and Storage Technologies

San Jose, CA USA February 13–16, 2007

conference proceedings

5th USENIX Conference on File and Storage Technologies

San Jose, CA, USA
February 13–16, 2007

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with ACM SIGOPS,
IEEE Mass Storage Systems Technical
Committee (MSSTC), and IEEE TCOS

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is $35 for members and $45 for nonmembers.
Outside the U.S.A. and Canada, please add $15 per copy for postage (via air printed matter).

Thanks to Our Sponsors

EMC^2
where information lives

vmware

hp
invent

IBM

NATIONAL SCIENCE FOUNDATION

SNIA

Thanks to Our Media Sponsors

Addison-Wesley Professional/
Prentice Hall Professional
ITtoolbox

Linux Journal
StorageNetworking.org

ISBN 1-931971-50-1

USENIX Association

Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)

February 13–16, 2007
San Jose, CA, USA

Symposium Organizers

Program Co-Chairs

Andrea C. Arpaci-Dusseau, *University of Wisconsin, Madison*
Remzi H. Arpaci-Dusseau, *University of Wisconsin, Madison*

Program Committee

Ashraf Aboulnaga, *University of Waterloo*
Mary Baker, *Hewlett-Packard Labs*
Bill Bolosky, *Microsoft*
Scott Brandt, *University of California, Santa Cruz*
Randal Burns, *Johns Hopkins University*
Mike Dahlin, *University of Texas, Austin*
Jason Flinn, *University of Michigan, Ann Arbor*
Dharmendra Modha, *IBM Almaden*
Erik Riedel, *Seagate*
M. Satyanarayanan, *Carnegie Mellon University*
Jiri Schindler, *Network Appliance*
Margo Seltzer, *Harvard University*
Kai Shen, *University of Rochester*
Anand Sivasubramaniam, *Pennsylvania State University*
Muthian Sivathanu, *Google*
Keith Smith, *Network Appliance*
Mike Swift, *University of Wisconsin, Madison*
Amin Vahdat, *University of California, San Diego*
Carl Waldspurger, *VMware*
Erez Zadok, *Stony Brook University*

The USENIX Association Staff

External Reviewers

Atul Adya
Nitin Agrawal
Guillermo Alvarez
Eric Anderson
James Anderson
Lakshmi Bairavasundaram
John Bent
Drew Bernat
Ricardo Bianchini
Angelo Bilas
Timothy Bisson
David Black
Elizabeth Borowsky
Nathan Burnett
Ali Butt
Enrique Carrera
Fay Chang
Jeff Chase
Shimin Chen
Zhifeng Chen
Tzi-cker Chiueh
Evan Cooke
Peter Corbett
Landon Cox
Timothy Denehy
David Dewitt
AnHai Doan
John Douceur
Fred Douglis
Allen Downey
Dan Ellard
Cristi Estan
Michael Feeley
Keir Fraser
Kevin Fu
Eran Gabber
Greg Ganger
Tal Garfinkel
Doug Ghormley
Garth Gibson
Jonathon Giffin
Richard Golding
Garth Goodson
Jim Gray
John Griffin
Dirk Grunwald
Haryadi Gunawi
Diwaker Gupta
Jim Haffner
Steve Hand
John Hartman
Val Henson
Peter Honeyman
Christopher Hoover
Jon Howell
Windsor Hsu
Sami Iren
Navendu Jain
Todd Jones
Tim Kaldewey
Mahesh Kallahalla
Michael Kaminsky
Kim Keeton
Emre Kiciman
Chip Killian
Eddie Kohler
Louis Kruger
Darrell Long
Jay Lorch
Xiaonan Ma
John MacCormick
Sam Madden
Petros Maniatis
J.P. Martin
Varun Marupadi
Jeanna Matthews
David Mazieres
Arif Merchant
Mike Mesnier
Ethan Miller
Klara Nahrstedt
Thu Nguyen
Ed Nightingale
Brian Noble
Vivek Pai
Shankar Pasupathy
Jignesh Patel
Dan Peek
Ina Popovici
Vijayan Prabhakaran
Raju Rangaswami
K.K. Rao
Sean Rhea
Alma Riska
Drew Roselli
Amir Roth
Mema Roussopolous
Steve Schlosser
Bianca Schroeder
Mehul Shah
Sam Shah
Prashant Shenoy
Gun Sirer
Craig Soules
Carl Staelin
John Strunk
Ya-Yunn Su
Ram Swaminathan
Nisha Talagala
Sandeep Tata
Doug Terry
Doug Thain
Eno Thereska
Sivan Toledo
Bhuvan Urgaonkar
Frank Uyeda
Mustafa Uysal
Kaushik Veeraraghavan
Catherine van Ingen
Rodney Van Meter
Peter Varman
Alistair Veitch
Ben Verghese
Harrick Vin
Yin Wang
Brent Welch
Matt Welsh
John Wilkes
Ted Wong
Jay Wylie
Lidong Zhang
Qingbo Zhu

5th USENIX Conference on File and Storage Technologies (FAST '07)

February 13–16, 2007
San Jose, CA, USA

Wednesday, February 14

Measure Thrice

Who Put Their Network in My Storage?

Thursday, February 15

The Latest Version

Thursday, February 15 (continued)

Scalable Systems

Cache Prizes

Friday, February 16

Beyond the Machine Room

Making the RAID

Index of Authors

Message from the Program Co-Chairs

Dear Colleagues,

It is with great pleasure that we welcome you to the 5th USENIX Conference on File and Storage Technologies (FAST '07). Historically, FAST has been a unique forum for research in storage systems, as it has successfully bridged the gap between exciting research ideas and best practices from industry. We believe that this year's program continues this fine tradition, and hope to see this rich area continue to flourish in the years to come.

FAST '07 received 98 interesting and broad-ranging submissions, and, in the end, the program committee decided to accept 19 excellent papers. Compared to previous FASTs, this FAST was right in the middle in terms of number of submissions, and as selective as the more selective FASTs:

Year	Accepted	Submitted	Percent
'02	21	110	19.1%
'03	18	67	26.9%
'04	18	72	25.0%
'05	25	125	20.0%
'07	19	98	19.4%

Of course, selectivity and similar metrics are not very accurate measures of the quality of a conference. Rather, one could look at the quality of the reviews of submitted papers. Here are some empirical statistics about the reviews submitted for this year's FAST:

The first three rows break down the papers into three groups: the accepted 19 papers ("considered and accepted"); the middle group of 59 papers, which had a realistic chance to get in but didn't ("considered but rejected"); and the bottom 20 papers ("not strongly considered"). Total numbers across all papers are in the bottommost row.

Which papers	Papers	Reviews	Total Words	Words/Paper
Considered and accepted	19	100	50,161	2640.1
Considered but rejected	59	307	162,656	2756.9
Not strongly considered	20	95	27,968	1398.4
All papers	98	502	240,785	2457.0

As one can see from the table, all papers received about 5 reviews (we assigned 3 to each PC member, and carefully selected 2 external reviewers for each). Those that were "considered" generally received substantial reviews, averaging over 2,600 words of reviews per paper. Not surprisingly, papers in the lowest group received noticeably shorter reviews; however, they still averaged nearly 1,400 words of feedback. Finally, although not statistically significant, one can see that papers that were considered but not accepted tended to receive the longest reviews; perhaps reviewers gave more feedback or justification (or both) when they felt a paper had merit but was likely to be rejected.

A different way to look at the review process is to marvel at the sheer quantity of commentary generated: almost a quarter of a million words. The program is comprised of 19 papers; when added together, these 19 papers consist of 192,193 words. With these figures in hand, one can compute the final input/output quotient for FAST: 1.25. Put simply, for every word of output in the proceedings, there was over a word of corresponding reviewer input.

Of course, there are many people to thank: all of those who submitted papers, particularly for enduring a last minute snafu that shut down the Web site an hour early and the resulting panic/heart attacks that ensued; all external reviewers (132 in total) for their wonderful and detailed commentaries; the program committee, for their preparation before the meeting, participation during the 2-day pre-OSDI festival where decisions were made, and shepherding of all accepted papers; and finally, to all the people at USENIX, the behind-the-scenes wizards of this (and other) conferences, who regularly transform the chaotic into the manageable and thus allow the community to focus on what is truly the heart of the process: reviewing the papers.

Thank you again for coming! We hope you have a wonderful FAST.

Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau, ***University of Wisconsin, Madison***
Program Co-Chairs

Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?

Bianca Schroeder   Garth A. Gibson
Computer Science Department
Carnegie Mellon University
{bianca, garth}@cs.cmu.edu

Abstract

Component failure in large-scale IT installations is becoming an ever larger problem as the number of components in a single cluster approaches a million.

In this paper, we present and analyze field-gathered disk replacement data from a number of large production systems, including high-performance computing sites and internet services sites. About 100,000 disks are covered by this data, some for an entire lifetime of five years. The data include drives with SCSI and FC, as well as SATA interfaces. The mean time to failure (MTTF) of those drives, as specified in their datasheets, ranges from 1,000,000 to 1,500,000 hours, suggesting a nominal annual failure rate of at most 0.88%.

We find that in the field, annual disk replacement rates typically exceed 1%, with 2-4% common and up to 13% observed on some systems. This suggests that field replacement is a fairly different process than one might predict based on datasheet MTTF.

We also find evidence, based on records of disk replacements in the field, that failure rate is not constant with age, and that, rather than a significant infant mortality effect, we see a significant early onset of wear-out degradation. That is, replacement rates in our data grew constantly with age, an effect often assumed not to set in until after a nominal lifetime of 5 years.

Interestingly, we observe little difference in replacement rates between SCSI, FC and SATA drives, potentially an indication that disk-independent factors, such as operating conditions, affect replacement rates more than component specific factors. On the other hand, we see only one instance of a customer rejecting an entire population of disks as a bad batch, in this case because of media error rates, and this instance involved SATA disks.

Time between replacement, a proxy for time between failure, is not well modeled by an exponential distribution and exhibits significant levels of correlation, including autocorrelation and long-range dependence.

1 Motivation

Despite major efforts, both in industry and in academia, high reliability remains a major challenge in running large-scale IT systems, and disaster prevention and cost of actual disasters make up a large fraction of the total cost of ownership. With ever larger server clusters, maintaining high levels of reliability and availability is a growing problem for many sites, including high-performance computing systems and internet service providers. A particularly big concern is the reliability of storage systems, for several reasons. First, failure of storage can not only cause temporary data unavailability, but in the worst case it can lead to permanent data loss. Second, technology trends and market forces may combine to make storage system failures occur more frequently in the future [24]. Finally, the size of storage systems in modern, large-scale IT installations has grown to an unprecedented scale with thousands of storage devices, making component failures the norm rather than the exception [7].

Large-scale IT systems, therefore, need better system design and management to cope with more frequent failures. One might expect increasing levels of redundancy designed for specific failure modes [3, 7], for example. Such designs and management systems are based on very simple models of component failure and repair processes [22]. Better knowledge about the statistical properties of storage failure processes, such as the distribution of time between failures, may empower researchers and designers to develop new, more reliable and available storage systems.

Unfortunately, many aspects of disk failures in real systems are not well understood, probably because the owners of such systems are reluctant to release failure data or do not gather such data. As a result, practitioners usually rely on vendor specified parameters, such as mean-time-to-failure (MTTF), to model failure processes, although many are skeptical of the accuracy of those models [4, 5, 33]. Too much academic and corporate research is based on anecdotes and back of the envelope calculations, rather than empirical data [28].

The work in this paper is part of a broader research agenda with the long-term goal of providing a better understanding of failures in IT systems by collecting, analyzing and making publicly available a diverse set of real failure histories from large-scale production systems. In our pursuit, we have spoken to a number of large production sites and were able to convince several of them to provide failure data from some of their systems.

In this paper, we provide an analysis of seven data sets we have collected, with a focus on storage-related failures. The data sets come from a number of large-scale production systems, including high-performance computing sites and large internet services sites, and consist primarily of hardware replacement logs. The data sets vary in duration from one month to five years and cover in total a population of more than 100,000 drives from at least four different vendors. Disks covered by this data include drives with SCSI and FC interfaces, commonly represented as the most reliable types of disk drives, as well as drives with SATA interfaces, common in desktop and nearline systems. Although 100,000 drives is a very large sample relative to previously published studies, it is small compared to the estimated 35 million enterprise drives, and 300 million total drives built in 2006 [1]. Phenomena such as bad batches caused by fabrication line changes may require much larger data sets to fully characterize.

We analyze three different aspects of the data. We begin in Section 3 by asking how disk replacement frequencies compare to replacement frequencies of other hardware components. In Section 4, we provide a quantitative analysis of disk replacement rates observed in the field and compare our observations with common predictors and models used by vendors. In Section 5, we analyze the statistical properties of disk replacement rates. We study correlations between disk replacements and identify the key properties of the empirical distribution of time between replacements, and compare our results to common models and assumptions. Section 6 provides an overview of related work and Section 7 concludes.

2 Methodology

2.1 What is a disk failure?

While it is often assumed that disk failures follow a simple fail-stop model (where disks either work perfectly or fail absolutely and in an easily detectable manner [22, 24]), disk failures are much more complex in reality. For example, disk drives can experience latent sector faults or transient performance problems. Often it is hard to correctly attribute the root cause of a problem to a particular hardware component.

Our work is based on hardware replacement records and logs, i.e. we focus on disk conditions that lead a drive customer to treat a disk as permanently failed and to replace it. We analyze records from a number of large production systems, which contain a record for every disk that was replaced in the system during the time of the data collection. To interpret the results of our work correctly it is crucial to understand the process of how this data was created. After a disk drive is identified as the likely culprit in a problem, the operations staff (or the computer system itself) perform a series of tests on the drive to assess its behavior. If the behavior qualifies as faulty according to the customer's definition, the disk is replaced and a corresponding entry is made in the hardware replacement log.

The important thing to note is that there is not one unique definition for when a drive is faulty. In particular, customers and vendors might use different definitions. For example, a common way for a customer to test a drive is to read all of its sectors to see if any reads experience problems, and decide that it is faulty if any one operation takes longer than a certain threshold. The outcome of such a test will depend on how the thresholds are chosen. Many sites follow a "better safe than sorry" mentality, and use even more rigorous testing. As a result, it cannot be ruled out that a customer may declare a disk faulty, while its manufacturer sees it as healthy. This also means that the definition of "faulty" that a drive customer uses does not necessarily fit the definition that a drive manufacturer uses to make drive reliability projections. In fact, a disk vendor has reported that for 43% of all disks returned by customers they find no problem with the disk [1].

It is also important to note that the failure behavior of a drive depends on the operating conditions, and not only on component level factors. For example, failure rates are affected by environmental factors, such as temperature and humidity, data center handling procedures, workloads and "duty cycles" or powered-on hours patterns.

We would also like to point out that the failure behavior of disk drives, even if they are of the same model, can differ, since disks are manufactured using processes and parts that may change. These changes, such as a change in a drive's firmware or a hardware component or even the assembly line on which a drive was manufactured, can change the failure behavior of a drive. This effect is often called the effect of batches or vintage. A bad batch can lead to unusually high drive failure rates or unusually high rates of media errors. For example, in the HPC3 data set (Table 1) the customer had 11,000 SATA drives replaced in Oct. 2006 after observing a high fre-

Data set	Type of cluster	Duration	#Disk events	# Servers	Disk Count	Disk Parameters	MTTF (Mhours)	Date of first Deploym.	ARR (%)
HPC1	HPC	08/01 - 05/06	474	765	2,318	18GB 10K SCSI	1.2	08/01	4.0
			124	64	1,088	36GB 10K SCSI	1.2		2.2
HPC2	HPC	01/04 - 07/06	14	256	520	36GB 10K SCSI	1.2	12/01	1.1
HPC3	HPC	12/05 - 11/06	103	1,532	3,064	146GB 15K SCSI	1.5	08/05	3.7
	HPC	12/05 - 11/06	4	N/A	144	73GB 15K SCSI	1.5		3.0
	HPC	12/05 - 08/06	253	N/A	11,000	250GB 7.2K SATA	1.0		3.3
HPC4	Various	09/03 - 08/06	269	N/A	8,430	250GB SATA	1.0	09/03	2.2
	HPC	11/05 - 08/06	7	N/A	2,030	500GB SATA	1.0	11/05	0.5
	clusters	09/05 - 08/06	9	N/A	3,158	400GB SATA	1.0	09/05	0.8
COM1	Int. serv.	May 2006	84	N/A	26,734	10K SCSI	1.0	2001	2.8
COM2	Int. serv.	09/04 - 04/06	506	9,232	39,039	15K SCSI	1.2	2004	3.1
COM3	Int. serv.	01/05 - 12/05	2	N/A	56	10K FC	1.2	N/A	3.6
			132	N/A	2,450	10K FC	1.2	N/A	5.4
			108	N/A	796	10K FC	1.2	N/A	13.6
			104	N/A	432	10K FC	1.2	1998	24.1

Table 1: *Overview of the seven failure data sets. Note that the disk count given in the table is the number of drives in the system at the end of the data collection period. For some systems the number of drives changed during the data collection period, and we account for that in our analysis. The disk parameters 10K and 15K refer to the rotation speed in revolutions per minute; drives not labeled 10K or 15K probably have a rotation speed of 7200 rpm.*

quency of media errors during writes. Although it took a year to resolve, the customer and vendor agreed that these drives did not meet warranty conditions. The cause was attributed to the breakdown of a lubricant leading to unacceptably high head flying heights. In the data, the replacements of these drives are not recorded as failures.

In our analysis we do not further study the effect of batches. We report on the field experience, in terms of disk replacement rates, of a set of drive customers. Customers usually do not have the information necessary to determine which of the drives they are using come from the same or different batches. Since our data spans a large number of drives (more than 100,000) and comes from a diverse set of customers and systems, we assume it also covers a diverse set of vendors, models and batches. We therefore deem it unlikely that our results are significantly skewed by "bad batches". However, we caution the reader not to assume all drives behave identically.

2.2 Specifying disk reliability and failure frequency

Drive manufacturers specify the reliability of their products in terms of two related metrics: the *annualized failure rate (AFR)*, which is the percentage of disk drives in a population that fail in a test scaled to a per year estimation; and the *mean time to failure (MTTF)*. The AFR of a new product is typically estimated based on accelerated life and stress tests or based on field data from earlier products [2]. The MTTF is estimated as the number of power on hours per year divided by the AFR. A common assumption for drives in servers is that they are powered on 100% of the time. Our data set providers all believe that their disks are powered on and in use at all times. The MTTFs specified for today's highest quality disks range from 1,000,000 hours to 1,500,000 hours, corresponding to AFRs of 0.58% to 0.88%. The AFR and MTTF estimates of the manufacturer are included in a drive's datasheet and we refer to them in the remainder as the *datasheet AFR* and the *datasheet MTTF*.

In contrast, in our data analysis we will report the *annual replacement rate (ARR)* to reflect the fact that, strictly speaking, disk replacements that are reported in the customer logs do not necessarily equal disk failures (as explained in Section 2.1).

2.3 Data sources

Table 1 provides an overview of the seven data sets used in this study. Data sets HPC1, HPC2 and HPC3 were collected in three large cluster systems at three different organizations using supercomputers. Data set HPC4 was collected on dozens of independently managed HPC sites, including supercomputing sites as well as commercial HPC sites. Data sets COM1, COM2, and COM3 were collected in at least three different cluster systems at a large internet service provider with many distributed and separately managed sites. In all cases, our data reports on only a portion of the computing systems run by each organization, as decided and selected by our sources.

It is important to note that for some systems the number of drives in the system changed significantly during the data collection period. While the table provides only the disk count at the end of the data collection period, our analysis in the remainder of the paper accounts for the actual date of these changes in the number of drives. Second, some logs also record events other than replacements, hence the number of disk events given in the table is not necessarily equal to the number of replacements or failures. The ARR values for the data sets can therefore not be directly computed from Table 1.

Below we describe each data set and the environment it comes from in more detail.

HPC1 is a five year log of hardware replacements collected from a 765 node high-performance computing cluster. Each of the 765 nodes is a 4-way SMP with 4 GB of memory and three to four 18GB 10K rpm SCSI drives. Of these nodes, 64 are used as filesystem nodes containing, in addition to the three to four 18GB drives, 17 36GB 10K rpm SCSI drives. The applications running on this system are typically large-scale scientific simulations or visualization applications. The data contains, for each hardware replacement that was recorded during the five year lifetime of this system, when the problem started, which node and which hardware component was affected, and a brief description of the corrective action.

HPC2 is a record of disk replacements observed on the compute nodes of a 256 node HPC cluster. Each node is a 4-way SMP with 16 GB of memory and contains two 36GB 10K rpm SCSI drives, except for eight of the nodes, which contain eight 36GB 10K rpm SCSI drives each. The applications running on this system are typically large-scale scientific simulations or visualization applications. For each disk replacement, the data set records the number of the affected node, the start time of the problem, and the slot number of the replaced drive.

HPC3 is a record of disk replacements observed on a 1,532 node HPC cluster. Each node is equipped with eight CPUs and 32GB of memory. Each node, except for four login nodes, has two 146GB 15K rpm SCSI disks. In addition, 11,000 7200 rpm 250GB SATA drives are used in an external shared filesystem and 144 73GB 15K rpm SCSI drives are used for the filesystem metadata. The applications running on this system are typically large-scale scientific simulations or visualization applications. For each disk replacement, the data set records the day of the replacement.

The HPC4 data set is a warranty service log of disk replacements. It covers three types of SATA drives used in dozens of separately managed HPC clusters. For the first type of drive, the data spans three years, for the other two types it spans a little less than a year. The data records, for each of the 13,618 drives, when it was first shipped and when (if ever) it was replaced in the field.

COM1 is a log of hardware failures recorded by an internet service provider and drawing from multiple distributed sites. Each record in the data contains a timestamp of when the failure was repaired, information on the failure symptoms, and a list of steps that were taken to diagnose and repair the problem. The data does not contain information on when each failure actually happened, only when repair took place. The data covers a population of 26,734 10K rpm SCSI disk drives. The total number of servers in the monitored sites is not known.

COM2 is a warranty service log of hardware failures recorded on behalf of an internet service provider aggregating events in multiple distributed sites. Each failure record contains a repair code (e.g. "Replace hard drive") and the time when the repair was finished. Again there is no information on the start time of each failure. The log does not contain entries for failures of disks that were replaced in the customer site by hot-swapping in a spare disk, since the data was created by the warranty processing, which does not participate in on-site hot-swap replacements. To account for the missing disk replacements we obtained numbers for the periodic replenishments of on-site spare disks from the internet service provider. The size of the underlying system changed significantly during the measurement period, starting with 420 servers in 2004 and ending with 9,232 servers in 2006. We obtained quarterly hardware purchase records covering this time period to estimate the size of the disk population in our ARR analysis.

The COM3 data set comes from a large external storage system used by an internet service provider and comprises four populations of different types of FC disks (see Table 1). While this data was gathered in 2005, the system has some legacy components that were as old as from 1998 and were known to have been physically moved after initial installation. We did not include these "obsolete" disk replacements in our analysis. COM3 differs from the other data sets in that it provides only aggregate statistics of disk failures, rather than individual records for each failure. The data contains the counts of disks that failed and were replaced in 2005 for each of the four disk populations.

2.4 Statistical methods

We characterize an empirical distribution using two import metrics: the mean and the squared coefficient of variation (C^2). The squared coefficient of variation is a measure of the variability of a distribution and is defined as the squared standard deviation divided by the squared mean. The advantage of using the squared coefficient of variation as a measure of variability, rather than the variance or the standard deviation, is that it is normalized by the mean, and so allows comparison of variability across distributions with different means.

We also consider the empirical cumulative distribution function (CDF) and how well it is fit by four probability distributions commonly used in reliability theory: the exponential distribution; the Weibull distribution; the gamma distribution; and the lognormal distribution. We parameterize the distributions through maximum likelihood estimation and evaluate the goodness of fit by visual inspection, the negative log-likelihood and the chi-square tests.

We will also discuss the hazard rate of the distribution of time between replacements. In general, the hazard rate of a random variable t with probability distribution $f(t)$ and cumulative distribution function $F(t)$ is defined as [25]

$$h(t) = \frac{f(t)}{1 - F(t)}$$

Intuitively, if the random variable t denotes the time between failures, the hazard rate $h(t)$ describes the instantaneous failure rate as a function of the time since the most recently observed failure. An important property of t's distribution is whether its hazard rate is constant (which is the case for an exponential distribution) or increasing or decreasing. A constant hazard rate implies that the probability of failure at a given point in time does not depend on how long it has been since the most recent failure. An increasing hazard rate means that the probability of a failure increases, if the time since the last failure has been long. A decreasing hazard rate means that the probability of a failure decreases, if the time since the last failure has been long.

The hazard rate is often studied for the distribution of lifetimes. It is important to note that we will focus on the hazard rate of the *time between disk replacements*, and not the hazard rate of disk lifetime distributions.

Since we are interested in correlations between disk failures we need a measure for the degree of correlation. The autocorrelation function (ACF) measures the correlation of a random variable with itself at different time lags l. The ACF, for example, can be used to determine whether the number of failures in one day is correlated with the number of failures observed l days later. The autocorrelation coefficient can range between 1 (high positive correlation) and -1 (high negative correlation). A value of zero would indicate no correlation, supporting independence of failures per day.

Another aspect of the failure process that we will study is long-range dependence. Long-range dependence measures the memory of a process, in particular how quickly the autocorrelation coefficient decays with growing lags. The strength of the long-range dependence is quantified by the Hurst exponent. A series exhibits long-range dependence if the Hurst exponent, H, is $0.5 < H < 1$. We use the Selfis tool [14] to obtain estimates of the Hurst parameter using five different methods: the absolute value method, the variance method, the R/S method, the periodogram method, and the Whittle estimator. A brief introduction to long-range dependence and a description of the Hurst parameter estimators is provided in [15].

HPC1	
Component	%
CPU	44
Memory	29
Hard drive	**16**
PCI motherboard	9
Power supply	2

Table 2: *Node outages that were attributed to hardware problems broken down by the responsible hardware component. This includes all outages, not only those that required replacement of a hardware component.*

3 Comparing disk replacement frequency with that of other hardware components

The reliability of a system depends on all its components, and not just the hard drive(s). A natural question is therefore what the relative frequency of drive failures is, compared to that of other types of hardware failures. To answer this question we consult data sets HPC1, COM1, and COM2, since these data sets contain records for all types of hardware replacements, not only disk replacements. Table 3 shows, for each data set, a list of the ten most frequently replaced hardware components and the fraction of replacements made up by each component. We observe that while the actual fraction of disk replacements varies across the data sets (ranging from 20% to 50%), it makes up a significant fraction in all three cases. In the HPC1 and COM2 data sets, disk drives are the most commonly replaced hardware component accounting for 30% and 50% of all hardware replacements, respectively. In the COM1 data set, disks are a close runner-up accounting for nearly 20% of all hardware replacements.

While Table 3 suggests that disks are among the most commonly replaced hardware components, it does not necessarily imply that disks are less reliable or have a shorter lifespan than other hardware components. The number of disks in the systems might simply be much larger than that of other hardware components. In order to compare the reliability of different hardware components, we need to normalize the number of component replacements by the component's population size.

Unfortunately, we do not have, for any of the systems, exact population counts of all hardware components. However, we do have enough information in HPC1 to estimate counts of the four most frequently replaced hard-

HPC1	
Component	%
Hard drive	**30.6**
Memory	28.5
Misc/Unk	14.4
CPU	12.4
PCI motherboard	4.9
Controller	2.9
QSW	1.7
Power supply	1.6
MLB	1.0
SCSI BP	0.3

COM1	
Component	%
Power supply	34.8
Memory	20.1
Hard drive	**18.1**
Case	11.4
Fan	8.0
CPU	2.0
SCSI Board	0.6
NIC Card	1.2
LV Power Board	0.6
CPU heatsink	0.6

COM2	
Component	%
Hard drive	**49.1**
Motherboard	23.4
Power supply	10.1
RAID card	4.1
Memory	3.4
SCSI cable	2.2
Fan	2.2
CPU	2.2
CD-ROM	0.6
Raid Controller	0.6

Table 3: *Relative frequency of hardware component replacements for the ten most frequently replaced components in systems HPC1, COM1 and COM2, respectively. Abbreviations are taken directly from service data and are not known to have identical definitions across data sets.*

ware components (CPU, memory, disks, motherboards). We estimate that there is a total of 3,060 CPUs, 3,060 memory dimms, and 765 motherboards, compared to a disk population of 3,406. Combining these numbers with the data in Table 3, we conclude that for the HPC1 system, the rate at which in five years of use a memory dimm was replaced is roughly comparable to that of a hard drive replacement; a CPU was about 2.5 times less often replaced than a hard drive; and a motherboard was 50% less often replaced than a hard drive.

The above discussion covers only failures that required a hardware component to be replaced. When running a large system one is often interested in any hardware failure that causes a node outage, not only those that necessitate a hardware replacement. We therefore obtained the HPC1 troubleshooting records for any node outage that was attributed to a hardware problem, including problems that required hardware replacements as well as problems that were fixed in some other way. Table 2 gives a breakdown of all records in the troubleshooting data, broken down by the hardware component that was identified as the root cause. We observe that 16% of all outage records pertain to disk drives (compared to 30% in Table 3), making it the third most common root cause reported in the data. The two most commonly reported outage root causes are CPU and memory, with 44% and 29%, respectively.

For a complete picture, we also need to take the severity of an anomalous event into account. A closer look at the HPC1 troubleshooting data reveals that a large number of the problems attributed to CPU and memory failures were triggered by parity errors, i.e. the number of errors is too large for the embedded error correcting code to correct them. In those cases, a simple reboot will bring the affected node back up. On the other hand, the majority of the problems that were attributed to hard disks (around 90%) lead to a drive replacement, which is a more expensive and time-consuming repair action.

Ideally, we would like to compare the frequency of hardware problems that we report above with the frequency of other types of problems, such software failures, network problems, etc. Unfortunately, we do not have this type of information for the systems in Table 1. However, in recent work [27] we have analyzed failure data covering any type of node outage, including those caused by hardware, software, network problems, environmental problems, or operator mistakes. The data was collected over a period of 9 years on more than 20 HPC clusters and contains detailed root cause information. We found that, for most HPC systems in this data, more than 50% of all outages are attributed to hardware problems and around 20% of all outages are attributed to software problems. Consistently with the data in Table 2, the two most common hardware components to cause a node outage are memory and CPU. The data of this recent study [27] is not used in this paper because it does not contain information about storage replacements.

4 Disk replacement rates

4.1 Disk replacements and MTTF

In the following, we study how field experience with disk replacements compares to datasheet specifications of disk reliability. Figure 1 shows the datasheet AFRs (horizontal solid and dashed line), the observed ARRs for each of the seven data sets and the weighted average ARR for all disks less than five years old (dotted line). For HPC1, HPC3, HPC4 and COM3, which cover different types of disks, the graph contains several bars, one for each type of disk, in the left-to-right order of the corresponding top-to-bottom entries in Table 1. Since at this

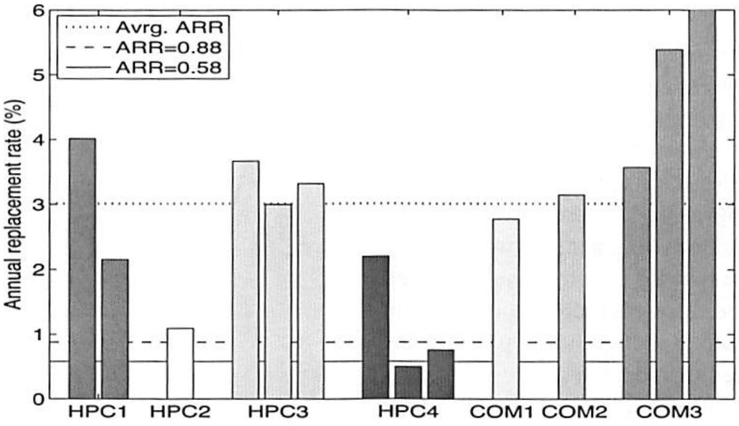


Figure 1: *Comparison of datasheet AFRs (solid and dashed line in the graph) and ARRs observed in the field. Each bar in the graph corresponds to one row in Table 1. The dotted line represents the weighted average over all data sets. Only disks within the nominal lifetime of five years are included, i.e. there is no bar for the COM3 drives that were deployed in 1998. The third bar for COM3 in the graph is cut off – its ARR is 13.5%.*

point we are not interested in wearout effects after the end of a disk's nominal lifetime, we have included in Figure 1 only data for drives within their nominal lifetime of five years. In particular, we do not include a bar for the fourth type of drives in COM3 (see Table 1), which were deployed in 1998 and were more than seven years old at the end of the data collection. These possibly "obsolete" disks experienced an ARR, during the measurement period, of 24%. Since these drives are well outside the vendor's nominal lifetime for disks, it is not surprising that the disks might be wearing out. All other drives were within their nominal lifetime and are included in the figure.

Figure 1 shows a significant discrepancy between the observed ARR and the datasheet AFR for all data sets. While the datasheet AFRs are between 0.58% and 0.88%, the observed ARRs range from 0.5% to as high as 13.5%. That is, the observed ARRs by data set and type, are by up to a factor of 15 higher than datasheet AFRs.

Most commonly, the observed ARR values are in the 3% range. For example, the data for HPC1, which covers almost exactly the entire nominal lifetime of five years exhibits an ARR of 3.4% (significantly higher than the datasheet AFR of 0.88%). The average ARR over all data sets (weighted by the number of drives in each data set) is 3.01%. Even after removing all COM3 data, which exhibits the highest ARRs, the average ARR was still 2.86%, 3.3 times higher than 0.88%.

It is interesting to observe that for these data sets there is no significant discrepancy between replacement rates for SCSI and FC drives, commonly represented as the most reliable types of disk drives, and SATA drives, frequently described as lower quality. For example, the ARRs of drives in the HPC4 data set, which are exclusively SATA drives, are among the lowest of all data sets. Moreover, the HPC3 data set includes both SCSI and SATA drives (as part of the same system in the same operating environment) and they have nearly identical replacement rates. Of course, these HPC3 SATA drives were decommissioned because of media error rates attributed to lubricant breakdown (recall Section 2.1), our only evidence of a bad batch, so perhaps more data is needed to better understand the impact of batches in overall quality.

It is also interesting to observe that the only drives that have an observed ARR below the datasheet AFR are the second and third type of drives in data set HPC4. One possible reason might be that these are relatively new drives, all less than one year old (recall Table 1). Also, these ARRs are based on only 16 replacements, perhaps too little data to draw a definitive conclusion.

A natural question arises: why are the observed disk replacement rates so much higher in the field data than the datasheet MTTF would suggest, even for drives in the first years of operation. As discussed in Sections 2.1 and 2.2, there are multiple possible reasons.

First, customers and vendors might not always agree on the definition of when a drive is "faulty". The fact that a disk was replaced implies that it failed some (possibly customer specific) health test. When a health test is conservative, it might lead to replacing a drive that the vendor tests would find to be healthy. Note, however, that even if we scale down the ARRs in Figure 1 to 57% of their actual values, to estimate the fraction of drives returned to the manufacturer that fail the latter's health test [1], the resulting AFR estimates are still more than a factor of two higher than datasheet AFRs in most cases.

Second, datasheet MTTFs are typically determined based on accelerated (stress) tests, which make certain assumptions about the operating conditions under which the disks will be used (e.g. that the temperature will always stay below some threshold), the workloads and "duty cycles" or powered-on hours patterns, and that certain data center handling procedures are followed. In practice, operating conditions might not always be as ideal as assumed in the tests used to determine datasheet MTTFs. A more detailed discussion of factors that can contribute to a gap between expected and measured drive reliability is given by Elerath and Shah [6].

Below we summarize the key observations of this section.

Observation 1: Variance between datasheet MTTF and disk replacement rates in the field was larger than we expected. The weighted average ARR was 3.4 times larger than 0.88%, corresponding to a datasheet MTTF of 1,000,000 hours.

Observation 2: For older systems (5-8 years of age), data sheet MTTFs underestimated replacement rates by as much as a factor of 30.

Observation 3: Even during the first few years of a system's lifetime (< 3 years), when wear-out is not expected to be a significant factor, the difference between datasheet MTTF and observed time to disk replacement was as large as a factor of 6.

Observation 4: In our data sets, the replacement rates of SATA disks are not worse than the replacement rates of SCSI or FC disks. This may indicate that disk-independent factors, such as operating conditions, usage and environmental factors, affect replacement rates more than component specific factors. However, the only evidence we have of a bad batch of disks was found in a collection of SATA disks experiencing high media error rates. We have too little data on bad batches to estimate the relative frequency of bad batches by type of disk, although there is plenty of anecdotal evidence that bad batches are not unique to SATA disks.

4.2 Age-dependent replacement rates

One aspect of disk failures that single-value metrics such as MTTF and AFR cannot capture is that in real life failure rates are not constant [5]. Failure rates of hardware products typically follow a "bathtub curve" with high failure rates at the beginning (infant mortality) and the end (wear-out) of the lifecycle. Figure 2 shows the failure rate pattern that is expected for the life cycle of hard drives [4, 5, 33]. According to this model, the first year of operation is characterized by early failures (or infant mortality). In years 2-5, the failure rates are approximately in steady state, and then, after years 5-7, wear-out starts to kick in.

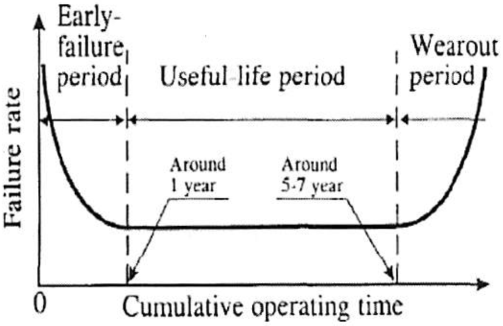


Figure 2: *Lifecycle failure pattern for hard drives [33].*

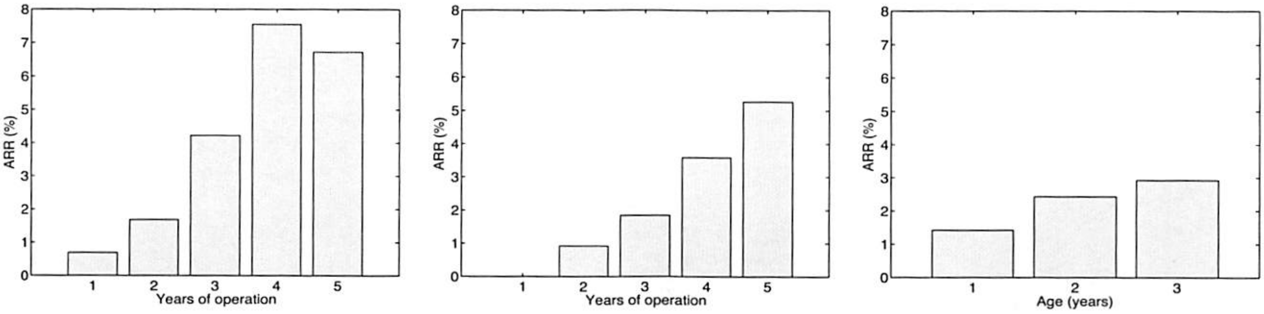
The common concern, that MTTFs do not capture infant mortality, has lead the International Disk drive Equipment and Materials Association (IDEMA) to propose a new standard for specifying disk drive reliability, based on the failure model depicted in Figure 2 [5, 33]. The new standard requests that vendors provide four different MTTF estimates, one for the first 1-3 months of operation, one for months 4-6, one for months 7-12, and one for months 13-60.

The goal of this section is to study, based on our field replacement data, how disk replacement rates in large-scale installations vary over a system's life cycle. Note that we only see customer visible replacement. Any infant mortality failure caught in the manufacturing, system integration or installation testing are probably not recorded in production replacement logs.

The best data sets to study replacement rates across the system life cycle are HPC1 and the first type of drives of HPC4. The reason is that these data sets span a long enough time period (5 and 3 years, respectively) and each cover a reasonably homogeneous hard drive population, allowing us to focus on the effect of age.

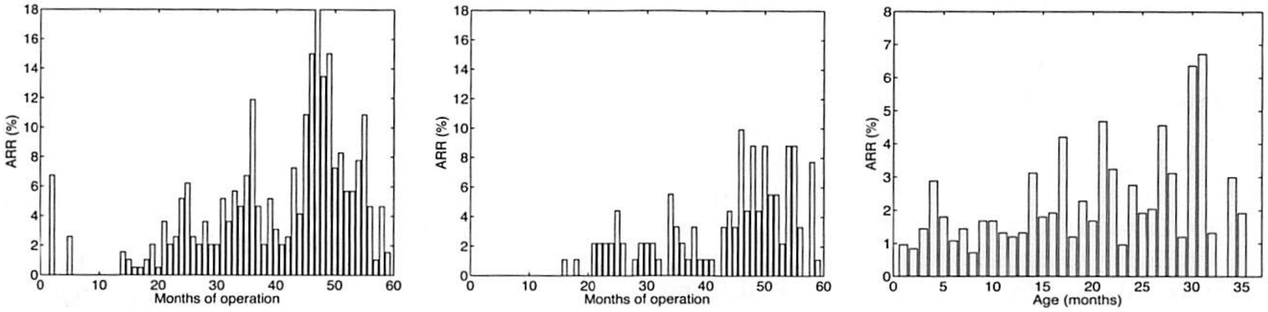
We study the change in replacement rates as a function of age at two different time granularities, on a per-month and a per-year basis, to make it easier to detect both short term and long term trends. Figure 3 shows the annual replacement rates for the disks in the compute nodes of system HPC1 (left), the file system nodes of system HPC1 (middle) and the first type of HPC4 drives (right), at a yearly granularity.

We make two interesting observations. First, replacement rates in all years, except for year 1, are larger than the datasheet MTTF would suggest. For example, in HPC1's second year, replacement rates are 20% larger than expected for the file system nodes, and a factor of



HPC1 (compute nodes) HPC1 (filesystem nodes) HPC4

Figure 3: *ARR for the first five years of system HPC1's lifetime, for the compute nodes (left) and the file system nodes (middle). ARR for the first type of drives in HPC4 as a function of drive age in years (right).*



HPC1 (compute nodes) HPC1 (filesystem nodes) HPC4

Figure 4: *ARR per month over the first five years of system HPC1's lifetime, for the compute nodes (left) and the file system nodes (middle). ARR for the first type of drives in HPC4 as a function of drive age in months (right).*

two larger than expected for the compute nodes. In year 4 and year 5 (which are still within the nominal lifetime of these disks), the actual replacement rates are 7–10 times higher than the failure rates we expected based on datasheet MTTF.

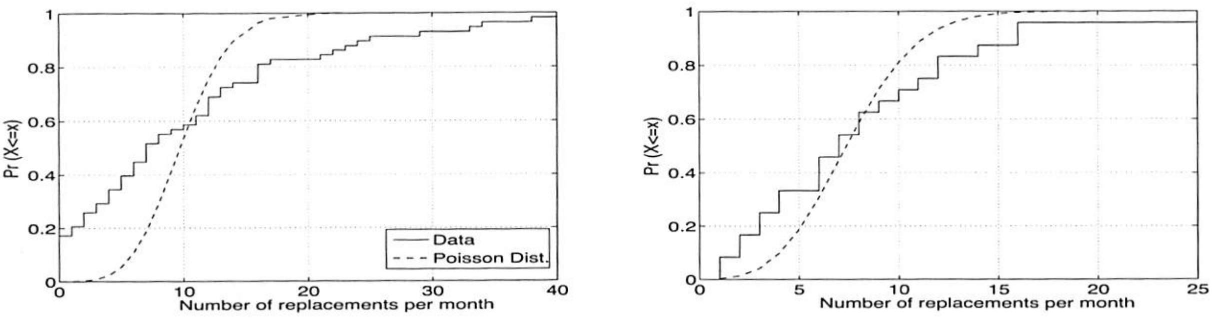
The second observation is that replacement rates are rising significantly over the years, even during early years in the lifecycle. Replacement rates in HPC1 nearly double from year 1 to 2, or from year 2 to 3. This observation suggests that wear-out may start much earlier than expected, leading to steadily increasing replacement rates during most of a system's useful life. This is an interesting observation because it does not agree with the common assumption that after the first year of operation, failure rates reach a steady state for a few years, forming the "bottom of the bathtub".

Next, we move to the per-month view of replacement rates, shown in Figure 4. We observe that for the HPC1 file system nodes there are no replacements during the first 12 months of operation, i.e. there's is no detectable infant mortality. For HPC4, the ARR of drives is not higher in the first few months of the first year than the last few months of the first year. In the case of the HPC1 compute nodes, infant mortality is limited to the first month of operation and is not above the steady state estimate of the datasheet MTTF. Looking at the lifecycle after month 12, we again see continuously rising replacement rates, instead of the expected "bottom of the bathtub".

Below we summarize the key observations of this section.

Observation 5: Contrary to common and proposed models, hard drive replacement rates do not enter steady state after the first year of operation. Instead replacement rates seem to steadily increase over time.

Observation 6: Early onset of wear-out seems to have a much stronger impact on lifecycle replacement rates than infant mortality, as experienced by end customers, even when considering only the first three or five years of a system's lifetime. We therefore recommend that wear-out be incorporated into new standards for disk drive reliability. The new standard suggested by IDEMA does not take wear-out into account [5, 33].



All years

Years 2-3

Figure 5: *CDF of number of disk replacements per month in HPC1*

5 Statistical properties of disk failures

In the previous sections, we have focused on aggregate statistics, e.g. the average number of disk replacements in a time period. Often one wants more information on the statistical properties of the time between failures than just the mean. For example, determining the expected time to failure for a RAID system requires an estimate on the probability of experiencing a second disk failure in a short period, that is while reconstructing lost data from redundant data. This probability depends on the underlying probability distribution and maybe poorly estimated by scaling an annual failure rate down to a few hours.

The most common assumption about the statistical characteristics of disk failures is that they form a Poisson process, which implies two key properties:

1. Failures are independent.
2. The time between failures follows an exponential distribution.

The goal of this section is to evaluate how realistic the above assumptions are. We begin by providing statistical evidence that disk failures in the real world are unlikely to follow a Poisson process. We then examine each of the two key properties (independent failures and exponential time between failures) independently and characterize in detail how and where the Poisson assumption breaks. In our study, we focus on the HPC1 data set, since this is the only data set that contains precise timestamps for when a problem was detected (rather than just timestamps for when repair took place).

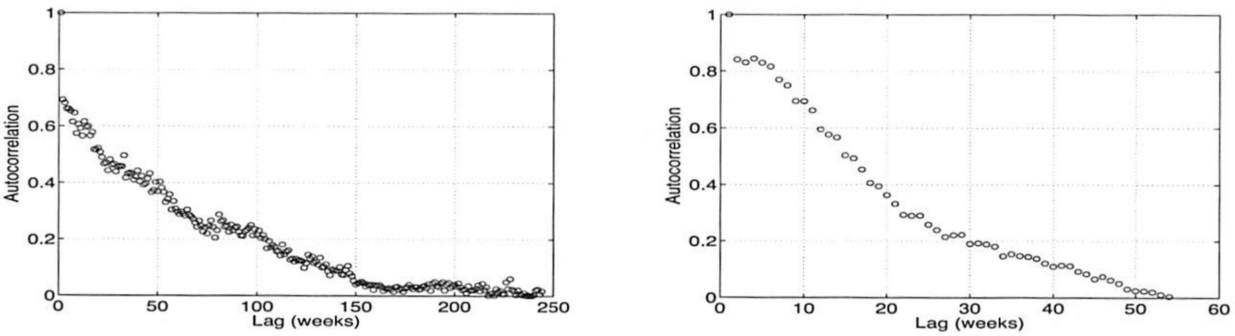
5.1 The Poisson assumption

The Poisson assumption implies that the number of failures during a given time interval (e.g. a week or a month) is distributed according to the Poisson distribution. Figure 5 (left) shows the empirical CDF of the number of disk replacements observed per month in the HPC1 data set, together with the Poisson distribution fit to the data's observed mean.

We find that the Poisson distribution does not provide a good visual fit for the number of disk replacements per month in the data, in particular for very small and very large numbers of replacements in a month. For example, under the Poisson distribution the probability of seeing ≥ 20 failures in a given month is less than 0.0024, yet we see 20 or more disk replacements in nearly 20% of all months in HPC1's lifetime. Similarly, the probability of seeing zero or one failure in a given month is only 0.0003 under the Poisson distribution, yet in 20% of all months in HPC1's lifetime we observe zero or one disk replacement.

A chi-square test reveals that we can reject the hypothesis that the number of disk replacements per month follows a Poisson distribution at the 0.05 significance level. All above results are similar when looking at the distribution of number of disk replacements per day or per week, rather than per month.

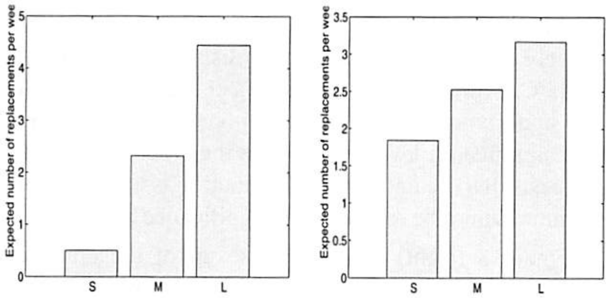
One reason for the poor fit of the Poisson distribution might be that failure rates are not steady over the lifetime of HPC1. We therefore repeat the same process for only part of HPC1's lifetime. Figure 5 (right) shows the distribution of disk replacements per month, using only data from years 2 and 3 of HPC1. The Poisson distribution achieves a better fit for this time period and the chi-square test cannot reject the Poisson hypothesis at a significance level of 0.05. Note, however, that this does not necessarily mean that the failure process during years 2 and 3 does follow a Poisson process, since this would also require the two key properties of a Poisson process (independent failures and exponential time between failures) to hold. We study these two properties in detail in the next two sections.



All years

Year 3

Figure 6: *Autocorrelation function for the number of disk replacements per week computed across the entire lifetime of the HPC1 system (left) and computed across only one year of HPC1's operation (right).*



All years

Year 3

Figure 7: *Expected number of disk replacements in a week depending on the number of disk replacements in the previous week.*

5.2 Correlations

In this section, we focus on the first key property of a Poisson process, the independence of failures. Intuitively, it is clear that in practice failures of disks in the same system are never completely independent. The failure probability of disks depends for example on many factors, such as environmental factors, like temperature, that are shared by all disks in the system. When the temperature in a machine room is far outside nominal values, all disks in the room experience a higher than normal probability of failure. The goal of this section is to statistically quantify and characterize the correlation between disk replacements.

We start with a simple test in which we determine the correlation of the number of disk replacements observed in successive weeks or months by computing the correlation coefficient between the number of replacements in a given week or month and the previous week or month. For data coming from a Poisson processes we would expect correlation coefficients to be close to 0. Instead we find significant levels of correlations, both at the monthly and the weekly level.

The correlation coefficient between consecutive weeks is 0.72, and the correlation coefficient between consecutive months is 0.79. Repeating the same test using only the data of one year at a time, we still find significant levels of correlation with correlation coefficients of 0.4-0.8.

Statistically, the above correlation coefficients indicate a strong correlation, but it would be nice to have a more intuitive interpretation of this result. One way of thinking of the correlation of failures is that the failure rate in one time interval is predictive of the failure rate in the following time interval. To test the strength of this prediction, we assign each week in HPC1's life to one of three buckets, depending on the number of disk replacements observed during that week, creating a bucket for weeks with small, medium, and large number of replacements, respectively [1]. The expectation is that a week that follows a week with a "small" number of disk replacements is more likely to see a small number of replacements, than a week that follows a week with a "large" number of replacements. However, if failures are independent, the number of replacements in a week will not depend on the number in a prior week.

Figure 7 (left) shows the expected number of disk replacements in a week of HPC1's lifetime as a function of which bucket the preceding week falls in. We observe that the expected number of disk replacements in a week varies by a factor of 9, depending on whether the preceding week falls into the first or third bucket, while we would expect no variation if failures were independent. When repeating the same process on the data of only year 3 of HPC1's lifetime, we see a difference of a close to factor of 2 between the first and third bucket.

So far, we have only considered correlations between successive time intervals, e.g. between two successive weeks. A more general way to characterize correlations is to study correlations at different time lags by using the autocorrelation function. Figure 6 (left) shows the autocorrelation function for the number of disk replacements

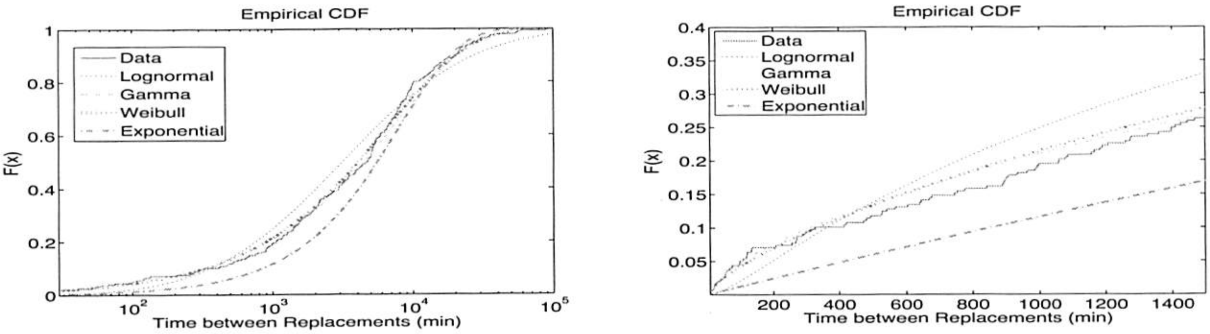


Figure 8: *Distribution of time between disk replacements across all nodes in HPC1.*

per week computed across the HPC1 data set. For a stationary failure process (e.g. data coming from a Poisson process) the autocorrelation would be close to zero at all lags. Instead, we observe strong autocorrelation even for large lags in the range of 100 weeks (nearly 2 years).

We repeated the same autocorrelation test for only parts of HPC1's lifetime and find similar levels of autocorrelation. Figure 6 (right), for example, shows the autocorrelation function computed only on the data of the third year of HPC1's life. Correlation is significant for lags in the range of up to 30 weeks.

Another measure for dependency is long range dependence, as quantified by the Hurst exponent H. The Hurst exponent measures how fast the autocorrelation functions drops with increasing lags. A Hurst parameter between 0.5–1 signifies a statistical process with a long memory and a slow drop of the autocorrelation function. Applying several different estimators (see Section 2) to the HPC1 data, we determine a Hurst exponent between 0.6-0.8 at the weekly granularity. These values are comparable to Hurst exponents reported for Ethernet traffic, which is known to exhibit strong long range dependence [16].

Observation 7: Disk replacement counts exhibit significant levels of autocorrelation.

Observation 8: Disk replacement counts exhibit long-range dependence.

5.3 Distribution of time between failure

In this section, we focus on the second key property of a Poisson failure process, the exponentially distributed time between failures. Figure 8 shows the empirical cumulative distribution function of time between disk replacements as observed in the HPC1 system and four distributions matched to it.

We find that visually the gamma and Weibull distributions are the best fit to the data, while exponential and lognormal distributions provide a poorer fit. This agrees with results we obtain from the negative log-likelihood, that indicate that the Weibull distribution is the best fit, closely followed by the gamma distribution. Performing a Chi-Square-Test, we can reject the hypothesis that the underlying distribution is exponential or lognormal at a significance level of 0.05. On the other hand the hypothesis that the underlying distribution is a Weibull or a gamma cannot be rejected at a significance level of 0.05.

Figure 8 (right) shows a close up of the empirical CDF and the distributions matched to it, for small time-between-replacement values (less than 24 hours). The reason that this area is particularly interesting is that a key application of the exponential assumption is in estimating the time until data loss in a RAID system. This time depends on the probability of a second disk failure during reconstruction, a process which typically lasts on the order of a few hours. The graph shows that the exponential distribution greatly underestimates the probability of a second failure during this time period. For example, the probability of seeing two drives in the cluster fail within one hour is four times larger under the real data, compared to the exponential distribution. The probability of seeing two drives in the cluster fail within the same 10 hours is two times larger under the real data, compared to the exponential distribution.

The poor fit of the exponential distribution might be due to the fact that failure rates change over the lifetime of the system, creating variability in the observed times between disk replacements that the exponential distribution cannot capture. We therefore repeated the above analysis considering only segments of HPC1's lifetime. Figure 9 shows as one example the results from analyzing the time between disk replacements in year 3 of HPC1's operation. While visually the exponential distribution now seems a slightly better fit, we can still reject the hypothesis of an underlying exponential distribution at a significance level of 0.05. The same holds for other 1-year and even 6-month segments of HPC1's lifetime. This leads us to believe that even during shorter segments

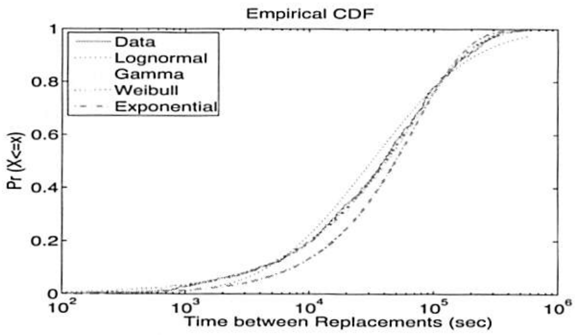


Figure 9: *Distribution of time between disk replacements across all nodes in HPC1 for only year 3 of operation.*

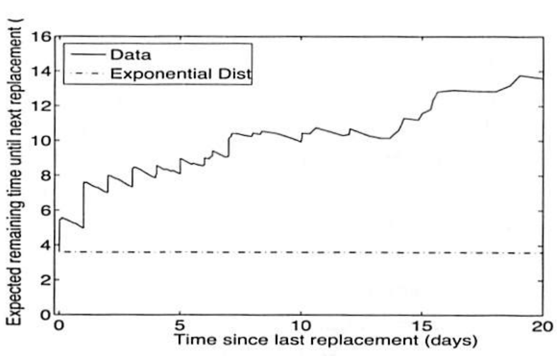


Figure 10: *Illustration of decreasing hazard rates*

of HPC1's lifetime the time between replacements is not realistically modeled by an exponential distribution.

While it might not come as a surprise that the simple exponential distribution does not provide as good a fit as the more flexible two-parameter distributions, an interesting question is what properties of the empirical time between failure make it different from a theoretical exponential distribution. We identify as a first differentiating feature that the data exhibits higher variability than a theoretical exponential distribution. The data has a C^2 of 2.4, which is more than two times higher than the C^2 of an exponential distribution, which is 1.

A second differentiating feature is that the time between disk replacements in the data exhibits decreasing hazard rates. Recall from Section 2.4 that the hazard rate function measures how the time since the last failure influences the expected time until the next failure. An increasing hazard rate function predicts that if the time since a failure is long then the next failure is coming soon. And a decreasing hazard rate function predicts the reverse. The table below summarizes the parameters for the Weibull and gamma distribution that provided the best fit to the data.

HPC1 Data	Distribution / Parameters			
	Weibull		Gamma	
	Shape	Scale	Shape	Scale
Compute nodes	0.73	0.037	0.65	176.4
Filesystem nodes	0.76	0.013	0.64	482.6
All nodes	0.71	0.049	0.59	160.9

Disk replacements in the filesystem nodes, as well as the compute nodes, and across all nodes, are fit best with gamma and Weibull distributions with a shape parameter less than 1, a clear indicator of decreasing hazard rates.

Figure 10 illustrates the decreasing hazard rates of the time between replacements by plotting the expected remaining time until the next disk replacement (Y-axis) as a function of the time since the last disk replacement (X-axis). We observe that right after a disk was replaced the expected time until the next disk replacement becomes necessary was around 4 days, both for the empirical data and the exponential distribution. In the case of the empirical data, after surviving for ten days without a disk replacement the expected remaining time until the next replacement had grown from initially 4 to 10 days; and after surviving for a total of 20 days without disk replacements the expected time until the next failure had grown to 15 days. In comparison, under an exponential distribution the expected remaining time stays constant (also known as the memoryless property).

Note, that the above result is not in contradiction with the increasing replacement rates we observed in Section 4.2 as a function of drive age, since here we look at the distribution of the time between disk replacements in a cluster, not disk lifetime distributions (i.e. how long did a drive live until it was replaced).

Observation 9: The hypothesis that time between disk replacements follows an exponential distribution can be rejected with high confidence.

Observation 10: The time between disk replacements has a higher variability than that of an exponential distribution.

Observation 11: The distribution of time between disk replacements exhibits decreasing hazard rates, that is, the expected remaining time until the next disk was replaced grows with the time it has been since the last disk replacement.

6 Related work

There is very little work published on analyzing failures in real, large-scale storage systems, probably as a result of the reluctance of the owners of such systems to release failure data.

Among the few existing studies is the work by Talagala et al. [29], which provides a study of error logs in a research prototype storage system used for a web server and includes a comparison of failure rates of different hardware components. They identify SCSI disk enclosures as the least reliable components and SCSI disks as one of the most reliable component, which differs from our results.

In a recently initiated effort, Schwarz et al. [28] have started to gather failure data at the Internet Archive, which they plan to use to study disk failure rates and bit rot rates and how they are affected by different environmental parameters. In their preliminary results, they report ARR values of 2–6% and note that the Internet Archive does not seem to see significant infant mortality. Both observations are in agreement with our findings.

Gray [31] reports the frequency of uncorrectable read errors in disks and finds that their numbers are smaller than vendor data sheets suggest. Gray also provides ARR estimates for SCSI and ATA disks, in the range of 3–6%, which is in the range of ARRs that we observe for SCSI drives in our data sets.

Pinheiro et al. analyze disk replacement data from a large population of serial and parallel ATA drives [23]. They report ARR values ranging from 1.7% to 8.6%, which agrees with our results. The focus of their study is on the correlation between various system parameters and drive failures. They find that while temperature and utilization exhibit much less correlation with failures than expected, the value of several SMART counters correlate highly with failures. For example, they report that after a scrub error drives are 39 times more likely to fail within 60 days than drives without scrub errors and that 44% of all failed drives had increased SMART counts in at least one of four specific counters.

Many have criticized the accuracy of MTTF based failure rate predictions and have pointed out the need for more realistic models. A particular concern is the fact that a single MTTF value cannot capture life cycle patterns [4, 5, 33]. Our analysis of life cycle patterns shows that this concern is justified, since we find failure rates to vary quite significantly over even the first two to three years of the life cycle. However, the most common life cycle concern in published research is underrepresenting infant mortality. Our analysis does not support this. Instead we observe significant underrepresentation of the early onset of wear-out.

Early work on RAID systems [8] provided some statistical analysis of time between disk failures for disks used in the 1980s, but didn't find sufficient evidence to reject the hypothesis of exponential times between failure with high confidence. However, time between failure has been analyzed for other, non-storage data in several studies [11, 17, 26, 27, 30, 32]. Four of the studies use distribution fitting and find the Weibull distribution to be a good fit [11, 17, 27, 32], which agrees with our results. All studies looked at the hazard rate function, but come to different conclusions. Four of them [11, 17, 27, 32] find decreasing hazard rates (Weibull shape parameter < 0.5). Others find that hazard rates are flat [30], or increasing [26]. We find decreasing hazard rates with Weibull shape parameter of 0.7-0.8.

Large-scale failure studies are scarce, even when considering IT systems in general and not just storage systems. Most existing studies are limited to only a few months of data, covering typically only a few hundred failures [13, 20, 21, 26, 30, 32]. Many of the most commonly cited studies on failure analysis stem from the late 80's and early 90's, when computer systems where significantly different from today [9, 10, 12, 17, 18, 19, 30].

7 Conclusion

Many have pointed out the need for a better understanding of what disk failures look like in the field. Yet hardly any published work exists that provides a large-scale study of disk failures in production systems. As a first step towards closing this gap, we have analyzed disk replacement data from a number of large production systems, spanning more than 100,000 drives from at least four different vendors, including drives with SCSI, FC and SATA interfaces. Below is a summary of a few of our results.

- Large-scale installation field usage appears to differ widely from nominal datasheet MTTF conditions. The field replacement rates of systems were significantly larger than we expected based on datasheet MTTFs.

- For drives less than five years old, field replacement rates were larger than what the datasheet MTTF suggested by a factor of 2–10. For five to eight year old drives, field replacement rates were a factor of 30 higher than what the datasheet MTTF suggested.

- Changes in disk replacement rates during the first five years of the lifecycle were more dramatic than often assumed. While replacement rates are often expected to be in steady state in year 2-5 of operation (bottom of the "bathtub curve"), we observed a continuous increase in replacement rates, starting as early as in the second year of operation.

- In our data sets, the replacement rates of SATA disks are not worse than the replacement rates of SCSI or FC disks. This may indicate that disk-independent factors, such as operating conditions, usage and environmental factors, affect replacement rates more than component specific factors. However, the only evidence we have of a bad batch of disks was found in a collection of SATA disks experiencing high media error rates. We have too little data on bad batches to estimate the relative frequency of bad batches by type of disk, although there is plenty of anecdotal evidence that bad batches are not unique to SATA disks.

- The common concern that MTTFs underrepresent infant mortality has led to the proposal of new standards that incorporate infant mortality [33]. Our findings suggest that the underrepresentation of the early onset of wear-out is a much more serious factor than underrepresentation of infant mortality and recommend to include this in new standards.

- While many have suspected that the commonly made assumption of exponentially distributed time between failures/replacements is not realistic, previous studies have not found enough evidence to prove this assumption wrong with significant statistical confidence [8]. Based on our data analysis, we are able to reject the hypothesis of exponentially distributed time between disk replacements with high confidence. We suggest that researchers and designers use field replacement data, when possible, or two parameter distributions, such as the Weibull distribution.

- We identify as the key features that distinguish the empirical distribution of time between disk replacements from the exponential distribution, higher levels of variability and decreasing hazard rates. We find that the empirical distributions are fit well by a Weibull distribution with a shape parameter between 0.7 and 0.8.

- We also present strong evidence for the existence of correlations between disk replacement interarrivals. In particular, the empirical data exhibits significant levels of autocorrelation and long-range dependence.

8 Acknowledgments

We would like to thank Jamez Nunez and Gary Grider from the High Performance Computing Division at Los Alamos National Lab and Katie Vargo, J. Ray Scott and Robin Flaus from the Pittsburgh Supercomputing Center for collecting and providing us with data and helping us to interpret the data. We also thank the other people and organizations, who have provided us with data, but would like to remain unnamed. For discussions relating to the use of high end systems, we would like to thank Mark Seager and Dave Fox of the Lawrence Livermore National Lab. Thanks go also to the anonymous reviewers and our shepherd, Mary Baker, for the many useful comments that helped improve the paper.

We thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Oracle, Panasas, Seagate, and Symantec) for their interest and support.

This material is based upon work supported by the Department of Energy under Award Number DE-FC02-06ER25767[2] and on research sponsored in part by the Army Research Office, under agreement number DAAD19–02–1–0389.

Notes

[1] More precisely, we choose the cutoffs between the buckets such that each bucket contains the same number of samples (i.e. weeks) by using the 33th percentile and the 66th percentile of the empirical distribution as cutoffs between the buckets.

[2] This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

[1] Personal communication with Dan Dummer, Andrei Khurshudov, Erik Riedel, Ron Watts of Seagate, 2006.

[2] G. Cole. Estimating drive reliability in desktop computers and consumer electronics systems. TP-338.1. Seagate. 2000.

[3] P. F. Corbett, R. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proc. of the FAST '04 Conference on File and Storage Technologies*, 2004.

[4] J. G. Elerath. AFR: problems of definition, calculation and measurement in a commercial environment. In *Proc. of the Annual Reliability and Maintainability Symposium*, 2000.

[5] J. G. Elerath. Specifying reliability in the disk drive industry: No more MTBFs. In *Proc. of the Annual Reliability and Maintainability Symposium*, 2000.

[6] J. G. Elerath and S. Shah. Server class drives: How reliable are they? In *Proc. of the Annual Reliability and Maintainability Symposium*, 2004.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[8] G. A. Gibson. Redundant disk arrays: Reliable, parallel secondary storage. Dissertation. MIT Press. 1992.

[9] J. Gray. Why do computers stop and what can be done about it. In *Proc. of the 5th Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[10] J. Gray. A census of tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4), 1990.

[11] T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. In *Proc. of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2002.

[12] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and modeling of computer reliability as affected by system activity. *ACM Trans. Comput. Syst.*, 4(3), 1986.

[13] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure data analysis of a LAN of Windows NT based computers. In *Proc. of the 18th IEEE Symposium on Reliable Distributed Systems*, 1999.

[14] T. Karagiannis. Selfis: A short tutorial. Technical report, University of California, Riverside, 2002.

[15] T. Karagiannis, M. Molle, and M. Faloutsos. Long-range dependence: Ten years of internet traffic modeling. *IEEE Internet Computing*, 08(5), 2004.

[16] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, 2(1), 1994.

[17] T.-T. Y. Lin and D. P. Siewiorek. Error log analysis: Statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, 39(4), 1990.

[18] J. Meyer and L. Wei. Analysis of workload influence on dependability. In *Proc. International Symposium on Fault-Tolerant Computing*, 1988.

[19] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International*, 11(5), 1995.

[20] D. Nurmi, J. Brevik, and R. Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. In *Euro-Par'05*, 2005.

[21] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[22] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 1988.

[23] E. Pinheiro, W. D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. of the FAST '07 Conference on File and Storage Technologies*, 2007.

[24] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, 2005.

[25] S. M. Ross. In *Introduction to probability models. 6th edition.* Academic Press.

[26] R. K. Sahoo, R. K., A. Sivasubramaniam, M. S. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proc. of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, 2004.

[27] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. In *Proc. of the 2006 International Conference on Dependable Systems and Networks (DSN'06)*, 2006.

[28] T. Schwarz, M. Baker, S. Bassi, B. Baumgart, W. Flagg, C. van Ingen, K. Joste, M. Manasse, and M. Shah. Disk failure investigations at the internet archive. In *Work-in-Progess session, NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST2006)*, 2006.

[29] N. Talagala and D. Patterson. An analysis of error behaviour in a large storage system. In *The IEEE Workshop on Fault Tolerance in Parallel and Distributed Systems*, 1999.

[30] D. Tang, R. K. Iyer, and S. S. Subramani. Failure analysis and modelling of a VAX cluster system. In *Proc. International Symposium on Fault-tolerant computing*, 1990.

[31] C. van Ingen and J. Gray. Empirical measurements of disk failure rates and error rates. In *MSR-TR-2005-166*, 2005.

[32] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT system field failure data analysis. In *Proc. of the 1999 Pacific Rim International Symposium on Dependable Computing*, 1999.

[33] J. Yang and F.-B. Sun. A comprehensive review of hard-disk drive reliability. In *Proc. of the Annual Reliability and Maintainability Symposium*, 1999.

Failure Trends in a Large Disk Drive Population

Eduardo Pinheiro, Wolf-Dietrich Weber and Luiz André Barroso
Google Inc.
1600 Amphitheatre Pkwy
Mountain View, CA 94043
{edpin,wolf,luiz}@google.com

Abstract

It is estimated that over 90% of all new information produced in the world is being stored on magnetic media, most of it on hard disk drives. Despite their importance, there is relatively little published work on the failure patterns of disk drives, and the key factors that affect their lifetime. Most available data are either based on extrapolation from accelerated aging experiments or from relatively modest sized field studies. Moreover, larger population studies rarely have the infrastructure in place to collect health signals from components in operation, which is critical information for detailed failure analysis.

We present data collected from detailed observations of a large disk drive population in a production Internet services deployment. The population observed is many times larger than that of previous studies. In addition to presenting failure statistics, we analyze the correlation between failures and several parameters generally believed to impact longevity.

Our analysis identifies several parameters from the drive's self monitoring facility (SMART) that correlate highly with failures. Despite this high correlation, we conclude that models based on SMART parameters alone are unlikely to be useful for predicting individual drive failures. Surprisingly, we found that temperature and activity levels were much less correlated with drive failures than previously reported.

1 Introduction

The tremendous advances in low-cost, high-capacity magnetic disk drives have been among the key factors helping establish a modern society that is deeply reliant on information technology. High-volume, consumer-grade disk drives have become such a successful product that their deployments range from home computers and appliances to large-scale server farms. In 2002, for example, it was estimated that over 90% of all new information produced was stored on magnetic media, most of it being hard disk drives [12]. It is therefore critical to improve our understanding of how robust these components are and what main factors are associated with failures. Such understanding can be particularly useful for guiding the design of storage systems as well as devising deployment and maintenance strategies.

Despite the importance of the subject, there are very few published studies on failure characteristics of disk drives. Most of the available information comes from the disk manufacturers themselves [2]. Their data are typically based on extrapolation from accelerated life test data of small populations or from returned unit databases. Accelerated life tests, although useful in providing insight into how some environmental factors can affect disk drive lifetime, have been known to be poor predictors of actual failure rates as seen by customers in the field [7]. Statistics from returned units are typically based on much larger populations, but since there is little or no visibility into the deployment characteristics, the analysis lacks valuable insight into what actually happened to the drive during operation. In addition, since units are typically returned during the warranty period (often three years or less), manufacturers' databases may not be as helpful for the study of long-term effects.

A few recent studies have shed some light on field failure behavior of disk drives [6, 7, 9, 16, 17, 19, 20]. However, these studies have either reported on relatively modest populations or did not monitor the disks closely enough during deployment to provide insights into the factors that might be associated with failures.

Disk drives are generally very reliable but they are also very complex components. This combination means that although they fail rarely, when they do fail, the possible causes of failure can be numerous. As a result, detailed studies of very large populations are the only way to collect enough failure statistics to enable meaningful conclusions. In this paper we present one such study by examining the population of hard drives under deployment within Google's computing infrastructure.

We have built an infrastructure that collects vital information about all Google's systems every few minutes, and a repository that stores these data in time-series format (essentially forever) for further analysis.

The information collected includes environmental factors (such as temperatures), activity levels and many of the Self-Monitoring Analysis and Reporting Technology (SMART) parameters that are believed to be good indicators of disk drive health. We mine through these data and attempt to find evidence that corroborates or contradicts many of the commonly held beliefs about how various factors can affect disk drive lifetime.

Our paper is unique in that it is based on data from a disk population size that is typically only available from vendor warranty databases, but has the depth of deployment visibility and detailed lifetime follow-up that only an end-user study can provide. Our key findings are:

- Contrary to previously reported results, we found very little correlation between failure rates and either elevated temperature or activity levels.
- Some SMART parameters (scan errors, reallocation counts, offline reallocation counts, and probational counts) have a large impact on failure probability.
- Given the lack of occurrence of predictive SMART signals on a large fraction of failed drives, it is unlikely that an accurate predictive failure model can be built based on these signals alone.

2 Background

In this section we describe the infrastructure that was used to gather and process the data used in this study, the types of disk drives included in the analysis, and information on how they are deployed.

2.1 The System Health Infrastructure

The System Health infrastructure is a large distributed software system that collects and stores hundreds of attribute-value pairs from all of Google's servers, and provides the interface for arbitrary analysis jobs to process that data.

The architecture of the System Health infrastructure is shown in Figure 1. It consists of a data collection layer, a distributed repository and an analysis framework. The collection layer is responsible for getting information from each of thousands of individual servers into a centralized repository. Different flavors of collectors exist to gather different types of data. Much of the health information is obtained from the machines directly. A daemon runs on every machine and gathers local data related to that machine's health, such as environmental parameters, utilization information of various resources, error indications, and configuration information. It is imperative that this daemon's resource usage be very light, so not to interfere with the applications. One way to assure this is to have the machine-level collector poll individual machines relatively infrequently (every few minutes). Other slower changing data (such as configuration information) and data from other existing databases can be collected even less frequently than that. Most notably for this study, data regarding machine repairs and disk swaps are pulled in from another database.

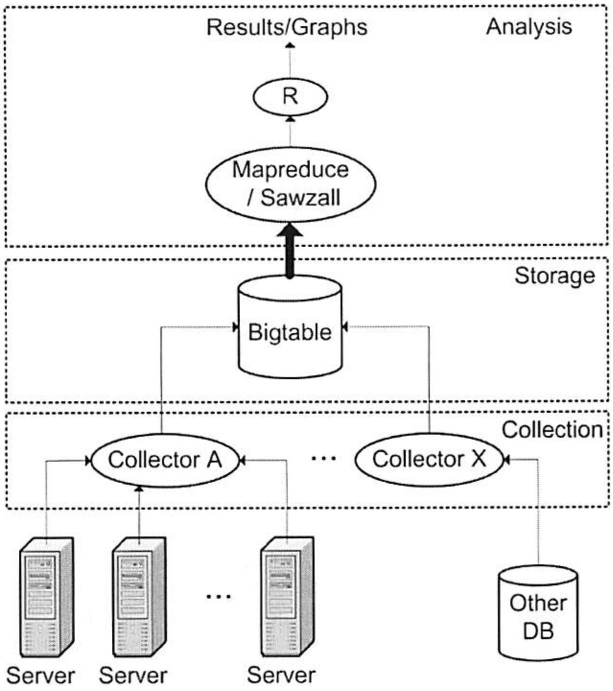


Figure 1: Collection, storage, and analysis architecture.

The System Health database is built upon Bigtable [3], a distributed data repository widely used within Google, which itself is built upon the Google File System (GFS) [8]. Bigtable takes care of all the data layout, compression, and access chores associated with a large data store. It presents the abstraction of a 2-dimensional table of data cells, with different versions over time making up a third dimension. It is a natural fit for keeping track of the values of different variables (columns) for different machines (rows) over time. The System Health database thus retains a complete time-ordered history of the environment, utilization, error, configuration, and repair events in each machine's life.

Analysis programs run on top of the System Health database, looking at information from individual machines, or mining the data across thousands of machines. Large-scale analysis programs are typically built upon Google's Mapreduce [5] framework. Mapreduce automates the mechanisms of large-scale distributed computation (such as work distribution, load balancing, tolerance of failures), allowing the user to focus simply on the algorithms that make up the heart of the computation.

The analysis pipeline used for this study consists of a Mapreduce job written in the Sawzall language and framework [15] to extract and clean up periodic SMART data and repair data related to disks, followed by a pass through R [1] for statistical analysis and final graph generation.

2.2 Deployment Details

The data in this study are collected from a large number of disk drives, deployed in several types of systems across all of Google's services. More than one hundred thousand disk drives were used for all the results presented here. The disks are a combination of serial and parallel ATA consumer-grade hard disk drives, ranging in speed from 5400 to 7200 rpm, and in size from 80 to 400 GB. All units in this study were put into production in or after 2001. The population contains several models from many of the largest disk drive manufacturers and from at least nine different models. The data used for this study were collected between December 2005 and August 2006.

As is common in server-class deployments, the disks were powered on, spinning, and generally in service for essentially all of their recorded life. They were deployed in rack-mounted servers and housed in professionally-managed datacenter facilities.

Before being put into production, all disk drives go through a short burn-in process, which consists of a combination of read/write stress tests designed to catch many of the most common assembly, configuration, or component-level problems. The data shown here do not include the fall-out from this phase, but instead begin when the systems are officially commissioned for use. Therefore our data should be consistent with what a regular end-user should see, since most equipment manufacturers put their systems through similar tests before shipment.

2.3 Data Preparation

Definition of Failure. Narrowly defining what constitutes a failure is a difficult task in such a large operation. Manufacturers and end-users often see different statistics when computing failures since they use different definitions for it. While drive manufacturers often quote yearly failure rates below 2% [2], user studies have seen rates as high as 6% [9]. Elerath and Shah [7] report between 15-60% of drives considered to have failed at the user site are found to have no defect by the manufacturers upon returning the unit. Hughes *et al.* [11] observe between 20-30% "no problem found" cases after analyzing failed drives from their study of 3477 disks.

From an end-user's perspective, a defective drive is one that misbehaves in a serious or consistent enough manner in the user's specific deployment scenario that it is no longer suitable for service. Since failures are sometimes the result of a combination of components (i.e., a particular drive with a particular controller or cable, etc), it is no surprise that a good number of drives that fail for a given user could be still considered operational in a different test harness. We have observed that phenomenon ourselves, including situations where a drive tester consistently "green lights" a unit that invariably fails in the field. Therefore, the most accurate definition we can present of a failure event for our study is: *a drive is considered to have failed if it was replaced as part of a repairs procedure*. Note that this definition implicitly excludes drives that were replaced due to an upgrade.

Since it is not always clear when exactly a drive failed, we consider the time of failure to be when the drive was replaced, which can sometimes be a few days after the observed failure event. It is also important to mention that the parameters we use in this study were not in use as part of the repairs diagnostics procedure at the time that these data were collected. Therefore there is no risk of false (forced) correlations between these signals and repair outcomes.

Filtering. With such a large number of units monitored over a long period of time, data integrity issues invariably show up. Information can be lost or corrupted along our collection pipeline. Therefore, some cleaning up of the data is necessary. In the case of missing values, the individual values are marked as not available and that specific piece of data is excluded from the detailed studies. Other records for that same drive are not discarded.

In cases where the data are clearly spurious, the entire record for the drive is removed, under the assumption that one piece of spurious data draws into question other fields for the same drive. Identifying spurious data, however, is a tricky task. Because part of the goal of studying the data is to learn what the numbers mean, we must be careful not to discard too much data that might appear invalid. So we define spurious simply as *negative counts or data values that are clearly impossible*. For example, some drives have reported temperatures that were hotter than the surface of the sun. Others have had negative power cycles. These were deemed spurious and removed. On the other hand, we have not filtered any suspiciously large counts from the SMART signals, under the hypothesis that large counts, while improbable as raw numbers, are likely to be good indicators of something really bad with the drive. Filtering for spurious values reduced the sample set size by less than 0.1%.

3 Results

We now analyze the failure behavior of our fleet of disk drives using detailed monitoring data collected over a nine-month observation window. During this time we recorded failure events as well as all the available environmental and activity data and most of the SMART parameters from the drives themselves. Failure information spanning a much longer interval (approximately five years) was also mined from an older repairs database. All the results presented here were tested for their statistical significance using the appropriate tests.

3.1 Baseline Failure Rates

Figure 2 presents the average Annualized Failure Rates (AFR) for all drives in our study, aged zero to 5 years, and is derived from our older repairs database. The data are broken down by the age a drive was when it failed. Note that this implies some overlap between the sample sets for the 3-month, 6-month, and 1-year ages, because a drive can reach its 3-month, 6-month and 1-year age all within the observation period. Beyond 1-year there is no more overlap.

While it may be tempting to read this graph as strictly failure rate with drive age, drive model factors are strongly mixed into these data as well. We tend to source a particular drive model only for a limited time (as new, more cost-effective models are constantly being introduced), so it is often the case that when we look at sets of drives of different ages we are also looking at a very different mix of models. Consequently, these data are not directly useful in understanding the effects of disk age on failure rates (the exception being the first three data points, which are dominated by a relatively stable mix of disk drive models). The graph is nevertheless a good way to provide a baseline characterization of failures across our population. It is also useful for later studies in the paper, where we can judge how consistent the impact of a given parameter is across these diverse drive model groups. A consistent and noticeable impact across all groups indicates strongly that the signal being measured has a fundamentally powerful correlation with failures, given that it is observed across widely varying ages and models.

The observed range of AFRs (see Figure 2) varies from 1.7%, for drives that were in their first year of operation, to over 8.6%, observed in the 3-year old population. The higher baseline AFR for 3 and 4 year old drives is more strongly influenced by the underlying reliability of the particular models in that vintage than by disk drive aging effects. It is interesting to note that our 3-month, 6-months and 1-year data points do seem to indicate a noticeable influence of infant mortality phenomena, with 1-year AFR dropping significantly from the AFR observed in the first three months.

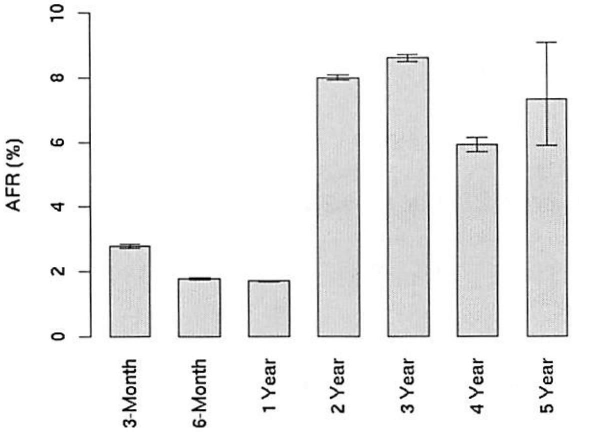


Figure 2: Annualized failure rates broken down by age groups

3.2 Manufacturers, Models, and Vintages

Failure rates are known to be highly correlated with drive models, manufacturers and vintages [18]. Our results do not contradict this fact. For example, Figure 2 changes significantly when we normalize failure rates per each drive model. Most age-related results are impacted by drive vintages. However, in this paper, we do not show a breakdown of drives per manufacturer, model, or vintage due to the proprietary nature of these data.

Interestingly, this does not change our conclusions. In contrast to age-related results, we note that all results shown in the rest of the paper are **not** affected significantly by the population mix. None of our SMART data results change significantly when normalized by drive model. The only exception is seek error rate, which is dependent on one specific drive manufacturer, as we discuss in section 3.5.5.

3.3 Utilization

The literature generally refers to utilization metrics by employing the term duty cycle which unfortunately has no consistent and precise definition, but can be roughly characterized as the fraction of time a drive is active out of the total powered-on time. What is widely reported in the literature is that higher duty cycles affect disk drives negatively [4, 21].

It is difficult for us to arrive at a meaningful numerical utilization metric given that our measurements do not provide enough detail to derive what 100% utilization might be for any given disk model. We choose instead to measure utilization in terms of weekly averages of read/write bandwidth per drive. We categorize utilization in three levels: low, medium and high, corresponding respectively to the lowest 25th percentile, 50-75th percentiles and top 75th percentile. This categorization is performed for each drive model, since the maximum bandwidths have significant variability across drive families. We note that using number of I/O operations and bytes transferred as utilization metrics provide very similar results. Figure 3 shows the impact of utilization on AFR across the different age groups.

Overall, we expected to notice a very strong and consistent correlation between high utilization and higher failure rates. However our results appear to paint a more complex picture. First, only very young and very old age groups appear to show the expected behavior. After the first year, the AFR of high utilization drives is at most moderately higher than that of low utilization drives. The three-year group in fact appears to have the opposite of the expected behavior, with low utilization drives having slightly higher failure rates than high utilization ones.

One possible explanation for this behavior is the *survival of the fittest* theory. It is possible that the failure modes that are associated with higher utilization are more prominent early in the drive's lifetime. If that is the case, the drives that survive the infant mortality phase are the least susceptible to that failure mode, and result in a population that is more robust with respect to variations in utilization levels.

Another possible explanation is that previous observations of high correlation between utilization and failures has been based on extrapolations from manufacturers' accelerated life experiments. Those experiments are likely to better model early life failure characteristics, and as such they agree with the trend we observe for the young age groups. It is possible, however, that longer term population studies could uncover a less pronounced effect later in a drive's lifetime.

When we look at these results across individual models we again see a complex pattern, with varying patterns of failure behavior across the three utilization levels. Taken as a whole, our data indicate a much weaker correlation between utilization levels and failures than previous work has suggested.

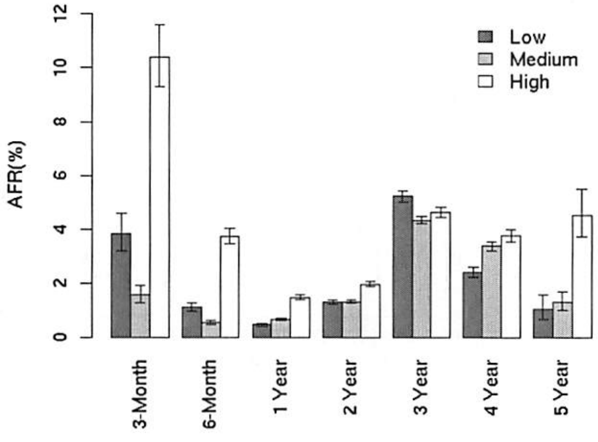


Figure 3: Utilization AFR

3.4 Temperature

Temperature is often quoted as the most important environmental factor affecting disk drive reliability. Previous studies have indicated that temperature deltas as low as 15C can nearly double disk drive failure rates [4]. Here we take temperature readings from the SMART records every few minutes during the entire 9-month window of observation and try to understand the correlation between temperature levels and failure rates.

We have aggregated temperature readings in several different ways, including averages, maxima, fraction of time spent above a given temperature value, number of times a temperature threshold is crossed, and last temperature before failure. Here we report data on averages and note that other aggregation forms have shown similar trends and and therefore suggest the same conclusions.

We first look at the correlation between average temperature during the observation period and failure. Figure 4 shows the distribution of drives with average temperature in increments of one degree and the corresponding annualized failure rates. The figure shows that failures do not increase when the average temperature increases. In fact, there is a clear trend showing that lower temperatures are associated with higher failure rates. Only at very high temperatures is there a slight reversal of this trend.

Figure 5 looks at the average temperatures for different age groups. The distributions are in sync with Figure 4 showing a mostly flat failure rate at mid-range temperatures and a modest increase at the low end of the temperature distribution. What stands out are the 3 and 4-year old drives, where the trend for higher failures with higher temperature is much more constant and also more pronounced.

Overall our experiments can confirm previously re-

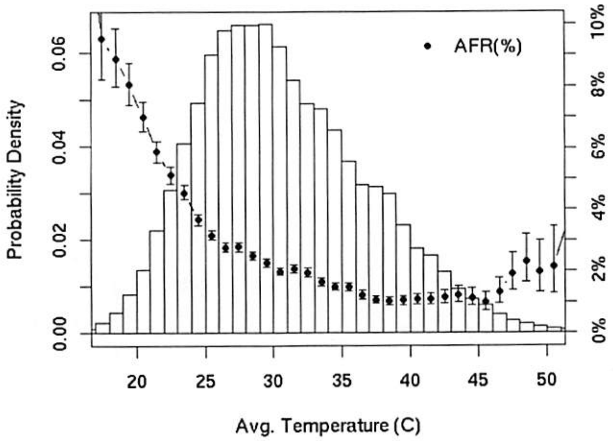


Figure 4: Distribution of average temperatures and failures rates.

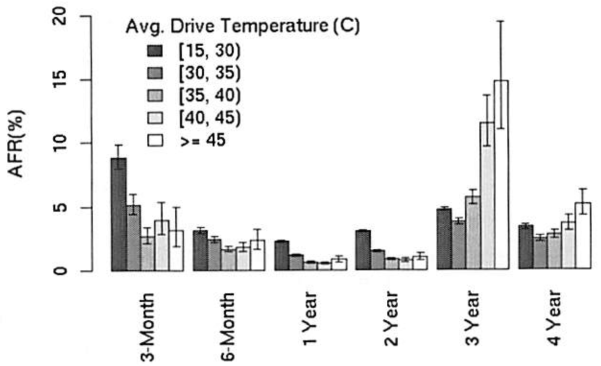


Figure 5: AFR for average drive temperature.

ported temperature effects only for the high end of our temperature range and especially for older drives. In the lower and middle temperature ranges, higher temperatures are not associated with higher failure rates. This is a fairly surprising result, which could indicate that datacenter or server designers have more freedom than previously thought when setting operating temperatures for equipment that contains disk drives. We can conclude that at moderate temperature ranges it is likely that there are other effects which affect failure rates much more strongly than temperatures do.

3.5 SMART Data Analysis

We now look at the various self-monitoring signals that are available from virtually all of our disk drives through the SMART standard interface. Our analysis indicates that some signals appear to be more relevant to the study of failures than others. We first look at those in detail, and then list a summary of our findings for the remaining ones. At the end of this section we discuss our results and reason about the usefulness of SMART parameters in obtaining predictive models for individual disk drive failures.

We present results in three forms. First we compare the AFR of drives with zero and non-zero counts for a given parameter, broken down by the same age groups as in figures 2 and 3. We also find it useful to plot the probability of survival of drives over the nine-month observation window for different ranges of parameter values. Finally, in addition to the graphs, we devise a single metric that could relay how relevant the values of a given SMART parameter are in predicting imminent failures. To that end, for each SMART parameter we look for thresholds that increased the probability of failure in the next 60 days by at least a factor of 10 with respect to drives that have zero counts for that parameter. We report such *Critical Thresholds* whenever we are able to find them with high confidence ($> 95\%$).

3.5.1 Scan Errors

Drives typically scan the disk surface in the background and report errors as they discover them. Large scan error counts can be indicative of surface defects, and therefore are believed to be indicative of lower reliability. In our population, fewer than 2% of the drives show scan errors and they are nearly uniformly spread across various disk models.

Figure 6 shows the AFR values of two groups of drives, those without scan errors and those with one or more. We plot bars across all age groups in which we have statistically significant data. We find that the group of drives with scan errors are ten times more likely to fail than the group with no errors. This effect is also noticed when we further break down the groups by disk model.

From Figure 8 we see a drastic and quick decrease in survival probability after the first scan error (left graph). A little over 70% of the drives survive the first 8 months after their first scan error. The dashed lines represent the 95% confidence interval. The middle plot in Figure 8 separates the population in four age groups (in months), and shows an effect that is not visible in the AFR plots. It appears that scan errors affect the survival probability of young drives more dramatically very soon after the first scan error occurs, but after the first month the curve flattens out. Older drives, however, continue to see a steady decline in survival probability throughout the 8-month period. This behavior could be another manifestation of infant mortality phenomenon. The right graph in figure 8 looks at the effect of multiple scan errors. While drives with one error are more likely to fail than those with none, drives with multiple errors fail even more quickly.

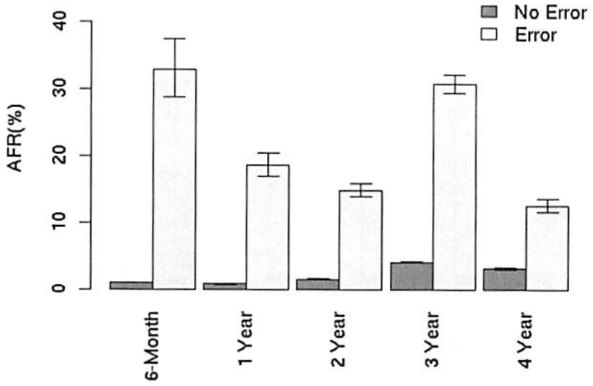


Figure 6: AFR for scan errors.

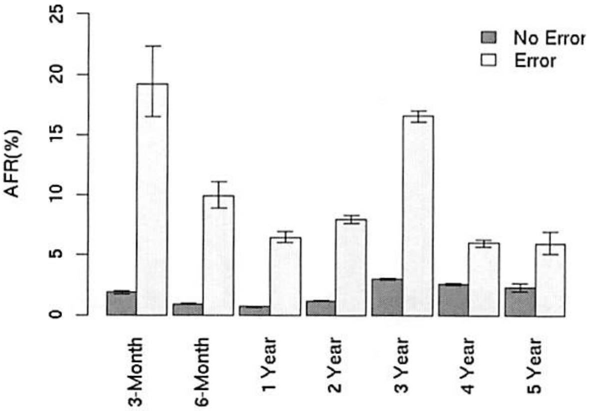


Figure 7: AFR for reallocation counts.

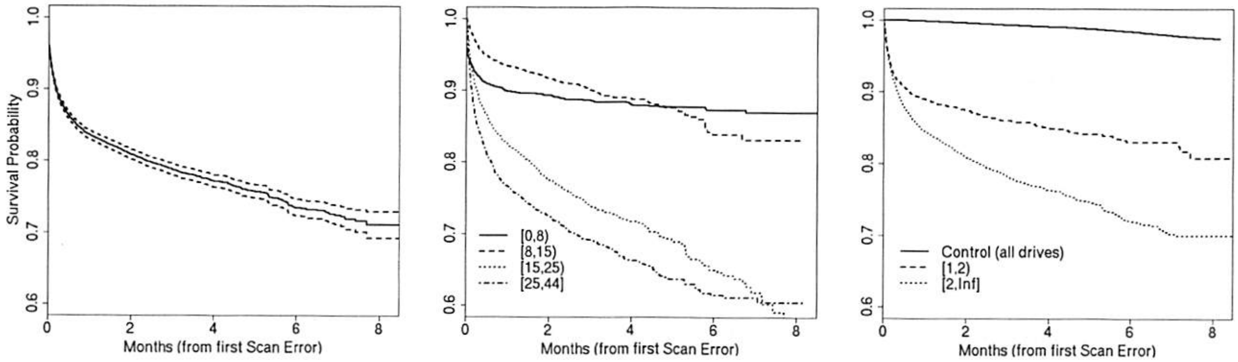


Figure 8: Impact of scan errors on survival probability. Left figure shows aggregate survival probability for all drives after first scan error. Middle figure breaks down survival probability per drive ages in months. Right figure breaks down drives by their number of scan errors.

The critical threshold analysis confirms what the charts visually imply: the critical threshold for scan errors is one. After the first scan error, drives are 39 times more likely to fail within 60 days than drives without scan errors.

3.5.2 Reallocation Counts

When the drive's logic believes that a sector is damaged (typically as a result of recurring soft errors or a hard error) it can remap the faulty sector number to a new physical sector drawn from a pool of spares. Reallocation counts reflect the number of times this has happened, and is seen as an indication of drive surface wear. About 9% of our population has reallocation counts greater than zero. Although some of our drive models show higher absolute values than others, the trends we observe are similar across all models.

As with scan errors, the presence of reallocations seems to have a consistent impact on AFR for all age groups (Figure 7), even if slightly less pronounced. Drives with one or more reallocations do fail more often than those with none. The average impact on AFR appears to be between a factor of 3-6x.

Figure 11 shows the survival probability after the first reallocation. We truncate the graph to 8.5 months, due to a drastic decrease in the confidence levels after that point. In general, the left graph shows, about 85% of the drives survive past 8 months after the first reallocation. The effect is more pronounced (middle graph) for drives in the age ranges [10,20) and [20, 60] months, while newer drives in the range [0,5) months suffer more than their next generation. This could again be due to infant mortality effects, although it appears to be less drastic in this case than for scan errors.

After their first reallocation, drives are over 14 times more likely to fail within 60 days than drives without reallocation counts, making the critical threshold for this parameter also one.

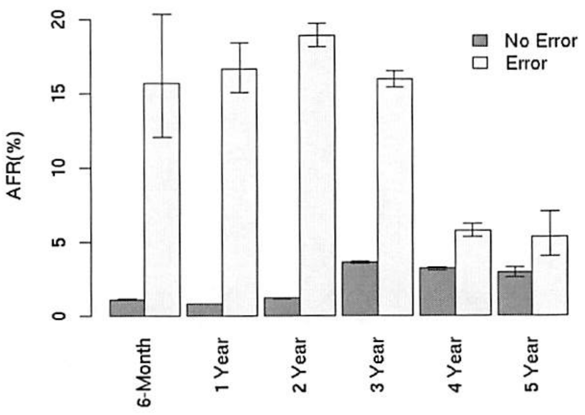


Figure 9: AFR for offline reallocation count.

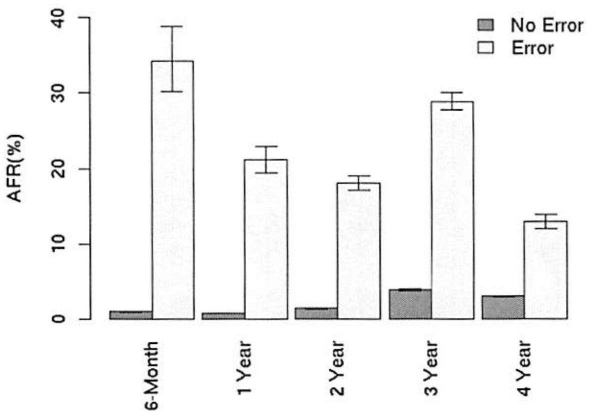


Figure 10: AFR for probational count.

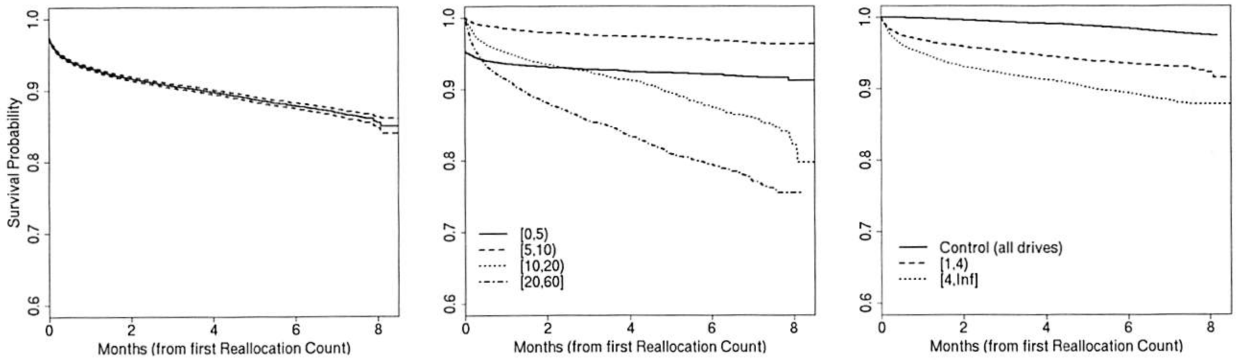


Figure 11: Impact of reallocation count values on survival probability. Left figure shows aggregate survival probability for all drives after first reallocation. Middle figure breaks down survival probability per drive ages in months. Right figure breaks down drives by their number of reallocations.

3.5.3 Offline Reallocations

Offline reallocations are defined as a subset of the reallocation counts studied previously, in which only reallocated sectors found during background scrubbing are counted. In other words, it should exclude sectors that are reallocated as a result of errors found during actual I/O operations. Although this definition mostly holds, we see evidence that certain disk models do not implement this definition. For instance, some models show more offline reallocations than total reallocations. Since the impact of offline reallocations appears to be significant and not identical to that of total reallocations, we decided to present it separately (Figure 9). About 4% of our population shows non-zero values for offline reallocations, and they tend to be concentrated on a particular subset of drive models.

Overall, the effects on survival probability of offline reallocation seem to be more drastic than those of total reallocations, as seen in Figure 12 (as before, some curves are clipped at 8 months because our data for those points were not within high confidence intervals). Drives in the older age groups appear to be more highly affected by it, although we are unable to attribute this effect to age given the different model mixes in the various age groups.

After the first offline reallocation, drives have over 21 times higher chances of failure within 60 days than drives without offline reallocations; an effect that is again more drastic than total reallocations.

Our data suggest that, although offline reallocations could be an important parameter affecting failures, it is particularly important to interpret trends in these values within specific models, since there is some evidence that different drive models may classify reallocations differently.

3.5.4 Probational Counts

Disk drives put suspect bad sectors "on probation" until they either fail permanently and are reallocated or continue to work without problems. Probational counts,

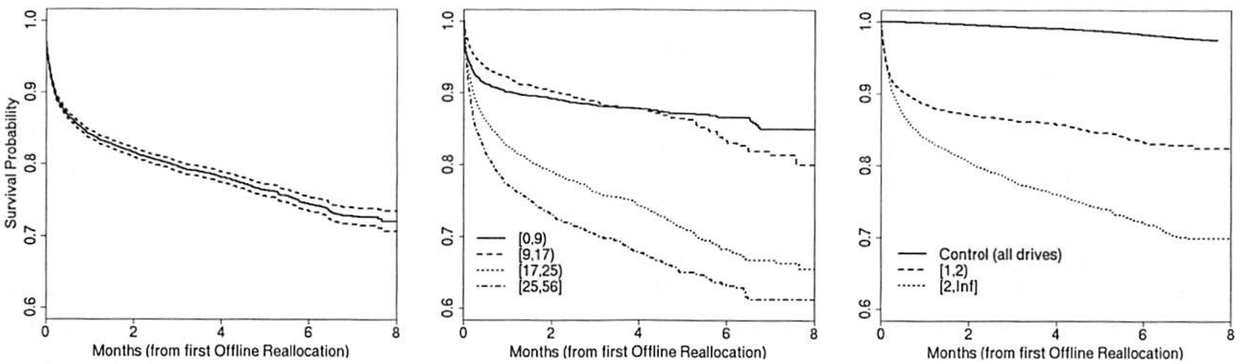


Figure 12: Impact of offline reallocation on survival probability. Left figure shows aggregate survival probability for all drives after first offline reallocation. Middle figure breaks down survival probability per drive ages in months. Right figure breaks down drives by their number offline reallocation.

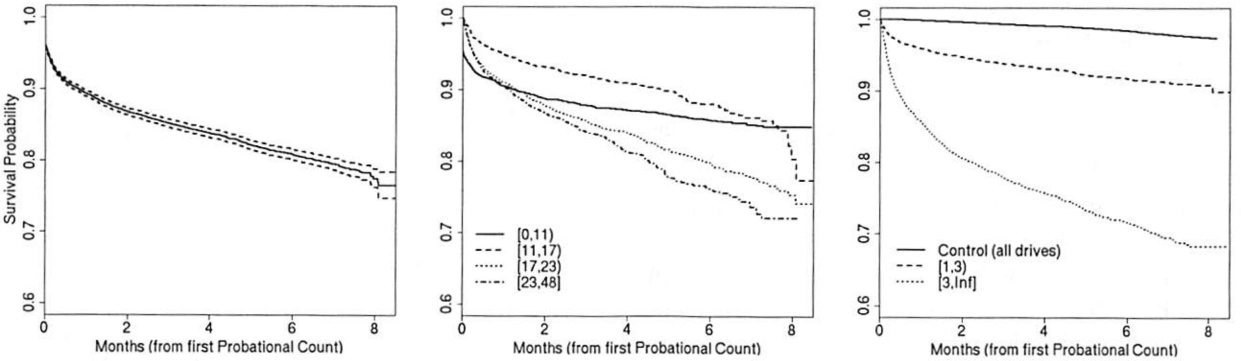


Figure 13: Impact of probational count values on survival probability. Left figure shows aggregate survival probability for all drives after first probational count. Middle figure breaks down survival probability per drive ages in months. Right figure breaks down drives by their number of probational counts.

therefore, can be seen as a softer error indication. It could provide earlier warning of possible problems but might also be a weaker signal, in that sectors on probation may indeed never be reallocated. About 2% of our drives had non-zero probational count values. We note that this number is lower than both online and offline reallocation counts, likely indicating that sectors may be removed from probation after further observation of their behavior. Once more, the distribution of drives with non-zero probational counts are somewhat skewed towards a subset of disk drive models.

Figures 10 and 13 show that probational count trends are generally similar to those observed for offline reallocations, with age group being somewhat less pronounced. The critical threshold for probational counts is also one: after the first event, drives are 16 times more likely to fail within 60 days than drives with zero probational counts.

3.5.5 Miscellaneous Signals

In addition to the SMART parameters described in the previous sections, which we have found to most closely impact failure rates, we have also studied several other parameters from the SMART set as well as other environmental factors. Here we briefly mention our relevant findings for some of those parameters.

Seek Errors. Seek errors occur when a disk drive fails to properly track a sector and needs to wait for another revolution to read or write from or to a sector. Drives report it as a rate, and it is meant to be used in combination with model-specific thresholds. When examining our population, we find that seek errors are widespread within drives of one manufacturer only, while others are more conservative in showing this kind of errors. For this one manufacturer, the trend in seek errors is not clear, changing from one vintage to another. For other manufacturers, there is no correlation between failure rates and seek errors.

CRC Errors. Cyclic redundancy check (CRC) errors are detected during data transmission between the physical media and the interface. Although we do observe some correlation between higher CRC counts and failures, those effects are somewhat less pronounced. CRC errors are less indicative of drive failures than that of cables and connectors. About 2% of our population had CRC errors.

Power Cycles. The power cycles indicator counts the number of times a drive is powered up and down. In a server-class deployment, in which drives are powered continuously, we do not expect to reach high enough power cycle counts to see any effects on failure rates. Our results find that for drives aged up to two years, this is true, there is no significant correlation between failures and high power cycles count. But for drives 3 years and older, higher power cycle counts can increase the absolute failure rate by over 2%. We believe this is due more to our population mix than to aging effects. Moreover, this correlation could be the effect (not the cause) of troubled machines that require many repair iterations and thus many power cycles to be fixed.

Calibration Retries. We were unable to reach a consistent and clear definition of this SMART parameter from public documents as well as consultations with some of the disk manufacturers. Nevertheless, our observations do not indicate that this is a particularly useful parameter for the goals of this study. Under 0.3% of our drives have calibration retries, and of that group only about 2% have failed, making this a very weak and imprecise signal when compared with other SMART parameters.

Spin Retries. Counts the number of retries when the drive is attempting to spin up. We did not register a single count within our entire population.

Power-on hours Although we do not dispute that power-on hours might have an effect on drive lifetime, it happens that in our deployment the age of the drive is an excellent approximation for that parameter, given that our drives remain powered on for most of their life time.

Vibration This is not a parameter that is part of the SMART set, but it is one that is of general concern in designing drive enclosures as most manufacturers describe how vibration can affect both performance and reliability of disk drives. Unfortunately we do not have sensor information to measure this effect directly for drives in service. We attempted to indirectly infer vibration effects by considering the differences in failure rates between systems with a single drive and those with multiple drives, but those experiments were not controlled enough for other possible factors to allow us to reach any conclusions.

3.5.6 Predictive Power of SMART Parameters

Given how strongly correlated some SMART parameters were found to be with higher failure rates, we were hopeful that accurate predictive failure models based on SMART signals could be created. Predictive models are very useful in that they can reduce service disruption due to failed components and allow for more efficient scheduled maintenance processes to replace the less efficient (and reactive) repairs procedures. In fact, one of the main motivations for SMART was to providé enough insight into disk drive behavior to enable such models to be built.

After our initial attempts to derive such models yielded relatively unimpressive results, we turned to the question of what might be the upper bound of the accuracy of any model based solely on SMART parameters. Our results are surprising, if not somewhat disappointing. Out of all failed drives, over 56% of them have no count in any of the four strong SMART signals, namely scan errors, reallocation count, offline reallocation, and probational count. In other words, models based only on those signals can never predict more than half of the failed drives. Figure 14 shows that even when we add all remaining SMART parameters (except temperature) we still find that over 36% of all failed drives had zero counts on all variables. This population includes seek error rates, which we have observed to be widespread in our population (> 72% of our drives have it) which further reduces the sample size of drives without any errors.

It is difficult to add temperature to this analysis since despite it being reported as part of SMART there are no crisp thresholds that directly indicate errors. However, if we arbitrarily assume that spending more than 50% of the observed time above 40C is an indication of possible problem, and add those drives to the set of predictable failures, we still are left with about 36% of all drives with no failure signals at all. Actual useful models, which need to have small false-positive rates are in fact likely to do much worse than these limits might suggest.

We conclude that it is unlikely that SMART data alone can be effectively used to build models that predict failures of individual drives. SMART parameters still appear to be useful in reasoning about the aggregate reliability of large disk populations, which is still very important for logistics and supply-chain planning. It is possible, however, that models that use parameters beyond those provided by SMART could achieve significantly better accuracies. For example, performance anomalies and other application or operating system signals could be useful in conjunction with SMART data to create more powerful models. We plan to explore this possibility in our future work.

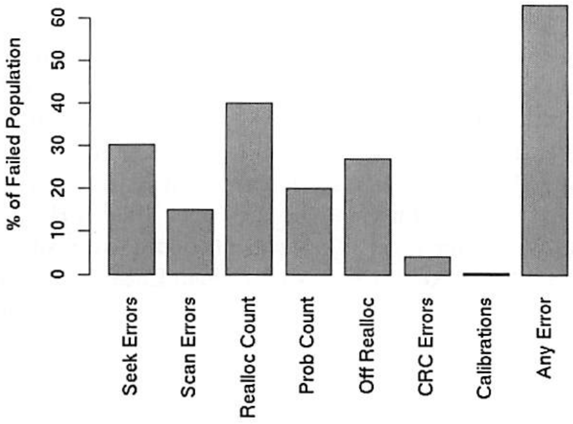


Figure 14: Percentage of failed drives with SMART errors.

4 Related Work

Previous studies in this area generally fall into two categories: vendor (disk drive or storage appliance) technical papers and user experience studies. Disk vendors studies provide valuable insight into the electromechanical characteristics of disks and both model-based and experimental data that suggests how several environmental factors and usage activities can affect device lifetime. Yang and Sun [21] and Cole [4] describe the processes and experimental setup used by Quantum and Seagate to test new units and the models that attempt to make long-term reliability predictions based on accelerated life tests of small populations. Power-on-hours, duty cycle, temperature are identified as the key deployment parameters that impact failure rates, each of them having the potential to double failure rates when going from nominal to extreme values. For example, Cole presents thermal de-rating models showing that MTBF could degrade by as much as 50% when going from operating temperatures of 30C to 40C. Cole's report also presents yearly failure rates from Seagate's warranty database, indicating a linear decrease in annual failure rates from 1.2% in the first year to 0.39% in the third (and last year of record). In our study, we did not find much correlation between failure rate and either elevated temperature or utilization. It is the most surprising result of our study. Our annualized failure rates were generally higher than those reported by vendors, and more consistent with other user experience studies.

Shah and Elerath have written several papers based on the behavior of disk drives inside Network Appliance storage products [6, 7, 19]. They use a reliability database that includes field failure statistics as well as support logs, and their position as an appliance vendor enables them more control and visibility into actual deployments than a typical disk drive vendor might have. Although they do not report directly on the correlation between SMART parameters or environmental factors and failures (possibly for confidentiality concerns), their work is useful in enabling a qualitative understanding of factors what affect disk drive reliability. For example, they comment that end-user failure rates can be as much as ten times higher than what the drive manufacturer might expect [7]; they report in [6] a strong experimental correlation between number of heads and higher failure rates (an effect that is also predicted by the models in [4]); and they observe that different failure mechanisms are at play at different phases of a drive lifetime [19]. Generally, our findings are in line with these results.

User experience studies may lack the depth of insight into the device inner workings that is possible in manufacturer reports, but they are essential in understanding device behavior in real-world deployments. Unfortunately, there are very few such studies to date, probably due to the large number of devices needed to observe statistically significant results and the complex infrastructure required to track failures and their contributing factors.

Talagala and Patterson [20] perform a detailed error analysis of 368 SCSI disk drives over an eighteen month period, reporting a failure rate of 1.9%. Results on a larger number of desktop-class ATA drives under deployment at the Internet Archive are presented by Schwarz et al [17]. They report on a 2% failure rate for a population of 2489 disks during 2005, while mentioning that replacement rates have been as high as 6% in the past. Gray and van Ingen [9] cite observed failure rates ranging from 3.3-6% in two large web properties with 22,400 and 15,805 disks respectively. A recent study by Schroeder and Gibson [16] helps shed light into the statistical properties of disk drive failures. The study uses failure data from several large scale deployments, including a large number of SATA drives. They report a significant overestimation of mean time to failure by manufacturers and a lack of infant mortality effects. None of these user studies have attempted to correlate failures with SMART parameters or other environmental factors.

We are aware of two groups that have attempted to correlate SMART parameters with failure statistics. Hughes et al [11, 13, 14] and Hamerly and Elkan [10]. The largest populations studied by these groups was of 3744 and 1934 drives and they derive failure models that achieve predictive rates as high as 30%, at false positive rates of about 0.2% (that false-positive rate corresponded to a number of drives between 20-43% of the drives that actually failed in their studies). Hughes *et al.*

also cites an annualized failure rate of 4-6%, based on their 2-3 month long experiment which appears to use stress test logs provided by a disk manufacturer.

Our study takes a next step towards a better understanding of disk drive failure characteristics by essentially combining some of the best characteristics of studies from vendor database analysis, namely population size, with the kind of visibility into a real-world deployment that is only possible with end-user data.

5 Conclusions

In this study we report on the failure characteristics of consumer-grade disk drives. To our knowledge, the study is unprecedented in that it uses a much larger population size than has been previously reported and presents a comprehensive analysis of the correlation between failures and several parameters that are believed to affect disk lifetime. Such analysis is made possible by a new highly parallel health data collection and analysis infrastructure, and by the sheer size of our computing deployment.

One of our key findings has been the lack of a consistent pattern of higher failure rates for higher temperature drives or for those drives at higher utilization levels. Such correlations have been repeatedly highlighted by previous studies, but we are unable to confirm them by observing our population. Although our data do not allow us to conclude that there is no such correlation, it provides strong evidence to suggest that other effects may be more prominent in affecting disk drive reliability in the context of a professionally managed data center deployment.

Our results confirm the findings of previous smaller population studies that suggest that some of the SMART parameters are well-correlated with higher failure probabilities. We find, for example, that after their first scan error, drives are 39 times more likely to fail within 60 days than drives with no such errors. First errors in reallocations, offline reallocations, and probational counts are also strongly correlated to higher failure probabilities. Despite those strong correlations, we find that failure prediction models based on SMART parameters alone are likely to be severely limited in their prediction accuracy, given that a large fraction of our failed drives have shown no SMART error signals whatsoever. This result suggests that SMART models are more useful in predicting trends for large aggregate populations than for individual components. It also suggests that powerful predictive models need to make use of signals beyond those provided by SMART.

Acknowledgments

We wish to acknowledge the contribution of numerous Google colleagues, particularly in the Platforms and Hardware Operations teams, who made this study possible, directly or indirectly; among them: Xiaobo Fan, Greg Slaughter, Don Yang, Jeremy Kubica, Jim Winget, Caio Villela, Justin Moore, Henry Green, Taliver Heath, and Walt Drummond. We are also thankful to our shepherd, Mary Baker for comments and guidance. A special thanks to Urs Hölzle for his extensive feedback on our drafts.

References

[1] The r project for statistical computing. http://www.r-project.org.

[2] Dave Anderson, Jim Dykes, and Erik Riedel. More than an interface - scsi vs. ata. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, pages 245 – 257, February 2003.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, November 2006.

[4] Gerry Cole. Estimating drive reliability in desktop computers and consumer electronics systems. *Seagate Technology Paper TP-338.1*, November 2000.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 137 – 150, December 2004.

[6] Jon G. Elerath and Sandeep Shah. Disk drive reliability case study: Dependence upon fly-height and quantity of heads. In *Proceedings of the Annual Symposium on Reliability and Maintainability*, January 2003.

[7] Jon G. Elerath and Sandeep Shah. Server class disk drives: How reliable are they? In *Proceedings of the Annual Symposium on Reliability and Maintainability*, pages 151 – 156, January 2004.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of*

the 19th ACM Symposium on Operating Systems Principles, pages 29 – 43, December 2003.

[9] Jim Gray and Catherine van Ingen. Empirical measurements of disk failure rates and error rates. *Technical Report MSR-TR-2005-166*, December 2005.

[10] Greg Hamerly and Charles Elkan. Bayesian approaches to failure prediction for disk drives. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML'01)*, June 2001.

[11] Gordon F. Hughes, Joseph F. Murray, Kenneth Kreutz-Delgado, and Charles Elkan. Improved disk-drive failure warnings. *IEEE Transactions on Reliability*, 51(3):350 – 357, September 2002.

[12] Peter Lyman and Hal R.Varian. How much information? October 2003. http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/index.htm.

[13] Joseph F. Murray, Gordon F Hughes, and Kenneth Kreutz-Delgado. Hard drive failure prediction using non-parametric statistical methods. *Proceedings of ICANN/ICONIP*, June 2003.

[14] Joseph F. Murray, Gordon F. Hughes, and Kenneth Kreutz-Delgado. Machine learning methods for predicting failures in hard drives: A multiple-instance application. *J. Mach. Learn. Res.*, 6:783–816, 2005.

[15] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, 13(4):227 – 298.

[16] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, February 2007.

[17] Thomas Schwartz, Mary Baker, Steven Bassi, Bruce Baumgart, Wayne Flagg, Catherine van Ingen, Kobus Joste, Mark Manasse, and Mehul Shah. Disk failure investigations at the internet archive. *14th NASA Goddard, 23rd IEEE Conference on Mass Storage Systems and Technologies*, May 2006.

[18] Sandeep Shah and Jon G. Elerath. Disk drive vintage and its effect on reliability. In *Proceedings of the Annual Symposium on Reliability and Maintainability*, pages 163 – 167, January 2004.

[19] Sandeep Shah and Jon G. Elerath. Reliability analysis of disk drive failure mechanisms. In *Proceedings of the Annual Symposium on Reliability and Maintainability*, pages 226 – 231, January 2005.

[20] Nisha Talagala and David Patterson. An analysis of error behavior in a large storage system. *Technical Report CSD-99-1042, University of California, Berkeley*, February 1999.

[21] Jimmy Yang and Feng-Bin Sun. A comprehensive review of hard-disk drive reliability. In *Proceedings of the Annual Symposium on Reliability and Maintainability*, pages 403 – 409, January 1999.

A Five-Year Study of File-System Metadata

Nitin Agrawal
University of Wisconsin, Madison
nitina@cs.wisc.edu

William J. Bolosky, John R. Douceur, Jacob R. Lorch
Microsoft Research
{bolosky,johndo,lorch}@microsoft.com

Abstract

For five years, we collected annual snapshots of file-system metadata from over 60,000 Windows PC file systems in a large corporation. In this paper, we use these snapshots to study temporal changes in file size, file age, file-type frequency, directory size, namespace structure, file-system population, storage capacity and consumption, and degree of file modification. We present a generative model that explains the namespace structure and the distribution of directory sizes. We find significant temporal trends relating to the popularity of certain file types, the origin of file content, the way the namespace is used, and the degree of variation among file systems, as well as more pedestrian changes in sizes and capacities. We give examples of consequent lessons for designers of file systems and related software.

1 Introduction

Every year from 2000 to 2004, we collected snapshots of metadata from over ten thousand file systems on the Windows desktop computers at Microsoft Corporation. We gathered this data by mass-emailing a scanning program to Microsoft's employees, and we had a 22% participation rate every year. Our resulting datasets contain metadata from 63,398 distinct file systems, 6457 of which provided snapshots in multiple years.

This project was a longitudinal extension of an earlier study we performed in 1998 [9], which was an order of magnitude larger than any prior study of file-system metadata. Our earlier study involved a single capture of file-system metadata, and it focused on lateral variation among file systems at a moment in time. By contrast, the present study focuses on longitudinal changes in file systems over a five-year time span.

In particular, we study temporal changes in the size, age, and type frequency of files; the size of directories; the structure of the file-system namespace; and various characteristics of file systems, including file and directory population, storage capacity, storage consumption, and degree of file modification.

The contributions of this work are threefold. First, we contribute the collected data set, which we will sanitize and make available for general use later this year. This is the largest set of file-system metadata ever collected, and it spans the longest time period of any sizeable metadata collection. To obtain this data set, contact the Microsoft authors.

Second, we contribute all of our research observations, including:

- The space used in file systems has increased over the course of our study, not only because mean file size has increased (from 108 KB to 189 KB), but also because the number of files has increased (from 30K to 90K).
- Eight file-name extensions account for over 35% of files, and nine file-name extensions account for over 35% of the bytes in files. The same sets of extensions have remained popular for many years.
- The fraction of file-system content created or modified locally has decreased over time. In the first year of our study, the median file system had 30% of its files created or modified locally, and four years later this percentage was 22%.
- Directory size distribution has not notably changed over the years of our study. In each year, directories have had very few subdirectories and a modest number of entries. 90% of them have had two or fewer subdirectories, and 90% of them have had 20 or fewer total entries.
- The fraction of file system storage residing in the namespace subtree meant for user documents and settings has increased in every year of our study, starting at 7% and rising to 15%. The fraction re-

siding in the subtree meant for system files has also risen over the course of our study, from 2% to 11%.

- File system capacity has increased dramatically during our study, with median capacity rising from 5 GB to 40 GB. One might expect this to cause drastic reductions in file system fullness, but instead the reduction in file system fullness has been modest. Median fullness has only decreased from 47% to 42%.
- Over the course of a single year, 80% of file systems become fuller and 18% become less full.

Third, we contribute a generative, probabilistic model for how directory trees are created. Our model explains the distribution of directories by depth in the namespace tree, and it also explains the distribution of the count of subdirectories per directory. This is the first generative model that characterizes the process by which file-system namespaces are constructed.

§2 describes the methodology of our data collection, analysis, and presentation. §3, §4, and §5 present our findings on, respectively, files, directories, and space usage. §6 surveys related work, and §7 summarizes and concludes.

2 Methodology

This section describes the methodology we applied to collecting, analyzing, and presenting the data.

2.1 Data collection

We developed a simple program that traverses the directory tree of each local, fixed-disk file system mounted on a computer. The program records a snapshot of all metadata associated with each file or directory, including hidden files and directories. This metadata includes name, size, timestamps, and attributes. The program also records the parent-child relationships of nodes in the namespace tree, as well as some system configuration information. The program records file names in an encrypted form. We wrote automated tools that decrypt the file names for computing aggregate statistics, but for privacy reasons we do not look at the decrypted file names directly, which places some limits on our analyses. In post-processing, we remove metadata relating to the system paging file, because this is part of the virtual memory system rather than the file system.

In the autumn of every year from 2000 to 2004, we distributed the scanning program via email to a large subset of the employees of Microsoft, with a request for the recipients to run the program on their desktop machines. As an incentive to participate, we held a lottery in which each scanned machine counted as an entry, with a single prize of a night's stay at a nearby resort hotel. The specific subset of people we were permitted to poll varied from year to year based on a number of factors; however, despite variations in user population and in other distribution particulars, we observed a 22% participation rate every year.

Year	Period	Users	Machs	FSs
2000	13 Sep – 29 Sep	5396	6051	11,654
2001	8 Oct – 2 Nov	7539	9363	16,022
2002	30 Sep – 1 Nov	7158	9091	15,011
2003	13 Oct – 14 Nov	7436	9262	14,633
2004	5 Oct – 12 Nov	7180	8729	13,505

Table 1: Properties of each year's dataset

Year	NTFS	FAT32	FAT	Other	Total
2000	7,015	2,696	1,943	0	11,654
2001	11,791	3,314	915	2	16,022
2002	12,302	2,280	429	0	15,011
2003	12,853	1,478	302	0	14,633
2004	12,364	876	264	1	13,505
Total	56,325	10,644	3,853	3	70,825

Table 2: File system types in datasets

We scanned desktops rather than servers because at Microsoft, files are typically stored on individual desktops rather than centralized servers. We collected the data via voluntary participation rather than random selection because the company only permitted the former approach; note that this voluntary approach may have produced selection bias.

2.2 Data properties

Table 1 itemizes some properties of each year's data collection. The primary collection period ran between the listed start and end dates, which mark the beginning of our emailing requests and the last eligible day for the lottery. Some snapshots continued to trickle in after the primary collection period; we used these in our analyses as well.

Table 2 itemizes the breakdown of each year's snapshots according to file-system type. 80% of our snapshots came from NTFS [27], the main file system for operating systems in the Windows NT family; 5% from FAT [18], a 16-bit file system dating from DOS; and 15% from FAT32 [18], a 32-bit upgrade of FAT developed for Windows 95.

Start	1	2	3	4	5
2000	11,654	950	234	63	18
2001	16,022	1,833	498	144	-
2002	15,011	1,852	588	-	-
2003	14,633	1,901	-	-	-
2004	13,505	-	-	-	-
Total	70,825	6,536	1,320	207	18

Table 3: Number of file systems for which we have snapshots in the n consecutive years starting with each year. For instance, there are 1,852 file systems for which we have snapshots from both 2002 and 2003.

For some analyses, we needed a way to establish whether two file-system snapshots from different years refer to the same file system. "Sameness" is not actually a well-formed notion; for example, it is not clear whether a file system is still the same after its volume is extended. We defined two snapshots to refer to the same file system if and only if they have the same user name, computer name, volume ID, drive letter, and total space. The need for some of these conditions was not obvious at first. For example, we added drive letter because some drives on some machines are multiply mapped, and we added total space so that a volume set would not be considered the same if a new volume were added to the set. Based on this definition, Table 3 shows the number of snapshots for which we have consecutive-year information.

2.3 Data presentation

Many of our graphs have horizontal axes that span a large range of nonnegative numbers. To represent these ranges compactly, we use a logarithmic scale for nonzero values, but we also include an abscissa for the zero value, even though zero does not strictly belong on a logarithmic scale.

We plot most histograms with line graphs rather than bar graphs because, with five or more datasets on a single plot, bar graphs can become difficult to read. For each bin in the histogram, we plot a point (x, y) where x is the midpoint of the bin and y is the size of the bin. We use the geometric midpoint when the x axis uses a logarithmic scale. We often plot un-normalized histograms rather than probability density functions (PDFs) for two reasons: First, the graphs expose more data if we do not normalize them. Second, because the count of files and directories per file system has grown substantially over time, not normalizing allows us to plot multiple years' curves on the same chart without overlapping to the point of unreadability.

Whenever we use the prefix K, as in KB, we mean 2^{10}. Similarly, we use M for 2^{20} and G for 2^{30}.

2.4 Data analysis

We believe that analysis of longitudinal file system data is of interest to many sets of people with diverse concerns about file system usage. For instance:

- developers of file systems, including desktop, server, and distributed file systems
- storage area network designers
- developers of file system utilities, such as backup, anti-virus, content indexing, encryption, and disk space usage visualization
- storage capacity planners
- disk manufacturers, especially those using gray-box techniques to enable visibility into the file system at the disk level [2]
- multitier storage system developers

In each subsection, after discussing our findings and what we consider to be the most interesting summaries of these findings, we will present some examples of interesting implications for the people enumerated above.

2.5 Limitations

All our data comes from a relatively homogenous sample of machines: Microsoft desktops running Windows. Since past studies [23, 28] have shown that file system characteristics can vary from one environment to another, our conclusions may not be applicable to substantially different environments. For instance, our conclusions are likely not applicable to file system server workloads, and it is unclear to what extent they can be generalized to non-Windows operating systems. It may also be that artifacts of Microsoft policy, such as specific software distributions that are common or disallowed, may yield results that would not apply to other workloads.

3 Files

3.1 File count per file system

Figure 1 plots cumulative distribution functions (CDFs) of file systems by count of files. The count of files per file system has increased steadily over our five-year sample period: The arithmetic mean has grown from 30K to 90K files and the median has grown from 18K to 52K files.

The count of files per file system is going up from year to year, and, as we will discuss in §4.1, the same holds

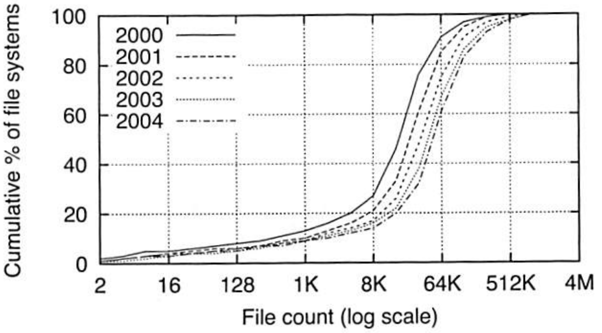


Figure 1: CDFs of file systems by file count

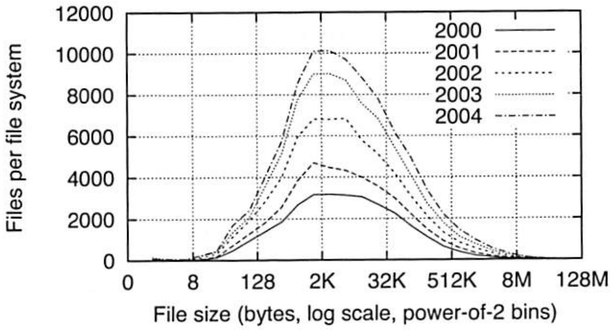


Figure 2: Histograms of files by size

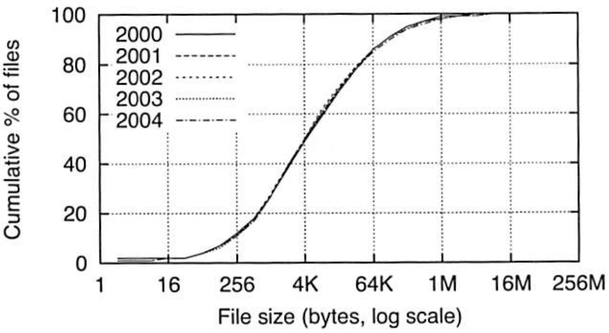


Figure 3: CDFs of files by size

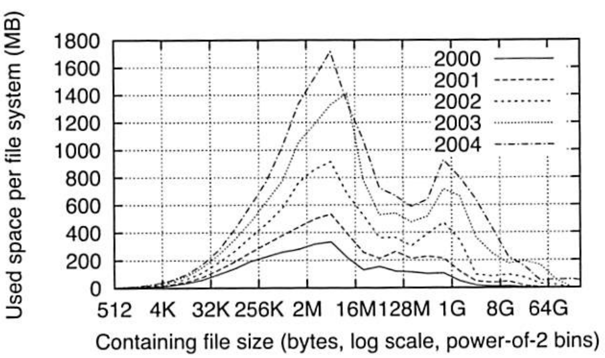


Figure 4: Histograms of bytes by containing file size

for directories. Thus, file system designers should ensure their metadata tables scale to large file counts. Additionally, we can expect file system scans that examine data proportional to the number of files and/or directories to take progressively longer. Examples of such scans include virus scans and metadata integrity checks following block corruption. Thus, it will become increasingly useful to perform these checks efficiently, perhaps by scanning in an order that minimizes movement of the disk arm.

3.2 File size

This section describes our findings regarding file size. We report the size of actual content, ignoring the effects of internal fragmentation, file metadata, and any other overhead. We observe that the overall file size distribution has changed slightly over the five years of our study. By contrast, the majority of stored bytes are found in increasingly larger files. Moreover, the latter distribution increasingly exhibits a double mode, due mainly to database and blob (binary large object) files.

Figure 2 plots histograms of files by size and Figure 3 plots the corresponding CDFs. We see that the absolute count of files per file system has grown significantly over time, but the general shape of the distribution has not changed significantly. Although it is not visible on the graph, the arithmetic mean file size has grown by 75% from 108 KB to 189 KB. In each year, 1–1.5% of files have a size of zero.

The growth in mean file size from 108 KB to 189 KB over four years suggests that this metric grows roughly 15% per year. Another way to estimate this growth rate is to compare our 2000 result to the 1981 result of 13.4 KB obtained by Satyanarayanan [24]. This comparison estimates the annual growth rate as 12%. Note that this latter estimate is somewhat flawed, since it compares file sizes from two rather different environments.

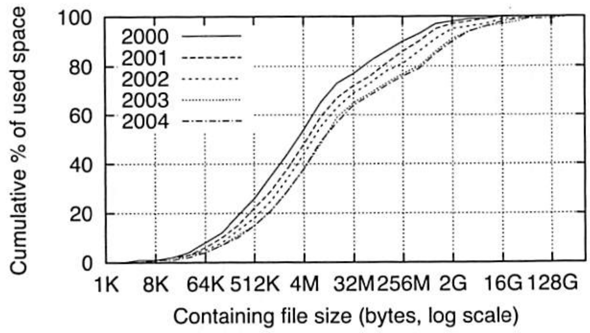


Figure 5: CDFs of bytes by containing file size

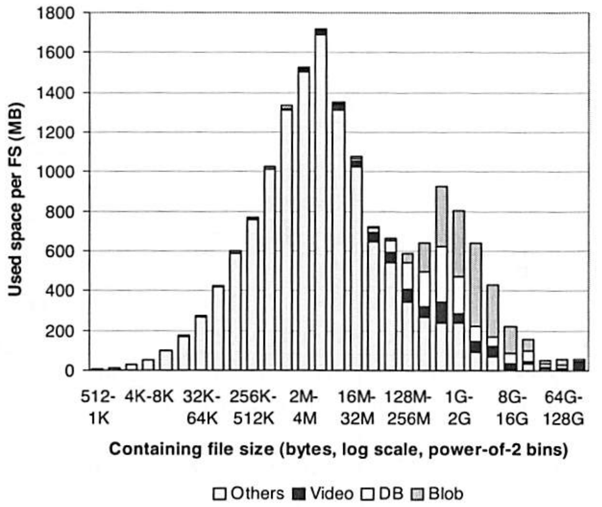


Figure 6: Contribution of file types to Figure 4 (2004). *Video* means files with extension `avi`, `dps`, `mpeg`, `mpg`, `vob`, or `wmv`; *DB* means files with extension `ldf`, `mad`, `mdf`, `ndf`, `ost`, or `pst`; and *Blob* means files named `hiberfil.sys` and files with extension `bak`, `bkf`, `bkp`, `dmp`, `gho`, `iso`, `pqi`, `rbf`, or `vhd`.

Figure 4 plots histograms of bytes by containing file size, alternately described as histograms of files weighted by file size. Figure 5 plots CDFs of these distributions. We observe that the distribution of file size has shifted to the right over time, with the median weighted file size increasing from 3 MB to 9 MB. Also, the distribution exhibits a double mode that has become progressively more pronounced. The corresponding distribution in our 1998 study did not show a true second mode, but it did show an inflection point around 64 MB, which is near the local minimum in Figure 4.

To study this second peak, we broke out several categories of files according to file-name extension. Figure 6 replots the 2004 data from Figure 4 as a stacked bar chart, with the contributions of video, database, and blob files indicated. We see that most of the bytes in large files are in video, database, and blob files, and that most of the video, database, and blob bytes are in large files.

Our finding that different types of files have different size distributions echoes the findings of other studies. In 1981, Satyanarayanan [24] found this to be the case on a shared file server in an academic environment. In 2001, Evans and Kuenning also noted this phenomenon in their analysis of 22 machines running various operating systems at Harvey Mudd College and Marine Biological Laboratories [11]. The fact that this finding is consistent across various different environments and times suggests that it is fundamental.

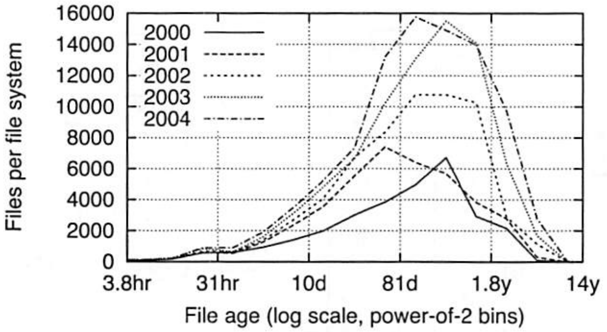


Figure 7: Histograms of files by age

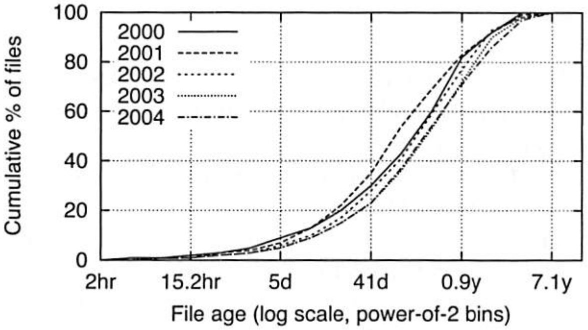


Figure 8: CDFs of files by age

There are several implications of the fact that a large number of small files account for a small fraction of disk usage, such as the following. First, it may not take much space to colocate many of these files with their metadata. This may be a reasonable way to reduce the disk seek time needed to access these files. Second, a file system that colocates several files in a single block, like ReiserFS [22], will have many opportunities to do so. This will save substantial space by eliminating internal fragmentation, especially if a large block size is used to improve performance. Third, designers of disk usage visualization utilities may want to show not only directories but also the names of certain large files.

3.3 File age

This subsection describes our findings regarding file age. Because file timestamps can be modified by application programs [17], our conclusions should be regarded cautiously.

Figure 7 plots histograms of files by age, calculated as the elapsed time since the file was created or last modified, relative to the time of the snapshot. Figure 8 shows CDFs of this same data. The median file age ranges between 80 and 160 days across datasets, with no clear trend over time.

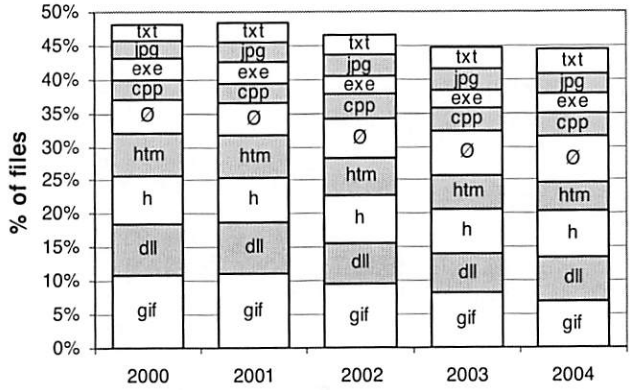


Figure 9: Fraction of files with popular extensions

The distribution of file age is not memoryless, so the age of a file is useful in predicting its remaining lifetime. So, systems such as archival backup systems can use this distribution to make predictions of how much longer a file will be needed based on how old it is. Since the distribution of file age has not appreciably changed across the years, we can expect that a prediction algorithm developed today based on the latest distribution will apply for several years to come.

3.4 File-name extensions

This subsection describes our findings regarding popular file types, as determined by file-name extension. Although the top few extensions have not changed dramatically over our five-year sample period, there has been some change, reflecting a decline in the relative prevalence of web content and an increase in use of virtual machines. The top few extensions account for nearly half of all files and bytes in file systems.

In old DOS systems with 8.3-style file names, the extension was the zero to three characters following the single dot in the file name. Although Windows systems allow file names of nearly arbitrary length and containing multiple dots, many applications continue to indicate their file types by means of extensions. For our analyses, we define an extension as the five-or-fewer characters following the last dot in a file name. If a name has no dots or has more than five characters after the last dot, we consider that name to have no extension, which we represent with the symbol Ø. As a special case, if a file name ends in `.gz`, `.bz2`, and `.Z`, then we ignore that suffix when determining extension. We do this because these are types of compressed files wherein the actual content type is indicated by the characters prior to the compression extension. To understand the typical usage of the file extensions we discuss in this section, see Table 4.

Extension	Typical Usage
`cpp`	C++ source code
`dll`	Dynamic link library
`exe`	Executable
`gif`	Image in Graphic Interchange Format
`h`	Source code header
`htm`	File in hypertext markup language
`jpg`	Image in JPEG format
`lib`	Code library
`mp3`	Music file in MPEG Layer III format
`pch`	Precompiled header
`pdb`	Source symbols for debugging
`pst`	Outlook personal folder
`txt`	Text
`vhd`	Virtual hard drive for virtual machine
`wma`	Windows Media Audio

Table 4: Typical usage of popular file extensions

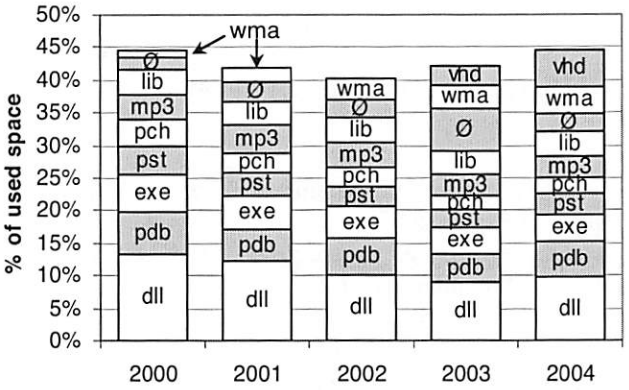


Figure 10: Fraction of bytes in files with popular extensions

Figure 9 plots, for the nine extensions that are the most popular in terms of file count, the fraction of files with that extension. The fractions are plotted longitudinally over our five-year sample period. The most notable thing we observe is that these extensions' popularity is relatively stable—the top five extensions have remained the top five for this entire time. However, the relative popularity of `gif` files and `htm` files has gone down steadily since 2001, suggesting a decline in the popularity of web content relative to other ways to fill one's file system.

Figure 10 plots, for the ten extensions that are the most popular in terms of summed file size, the fraction of file bytes residing in files with that extension. Across all years, dynamic link libraries (`dll` files) contain more bytes than any other file type. Extension `vhd`, which is used for virtual hard drives, is consuming a rapidly increasing fraction of file-system space, suggesting that

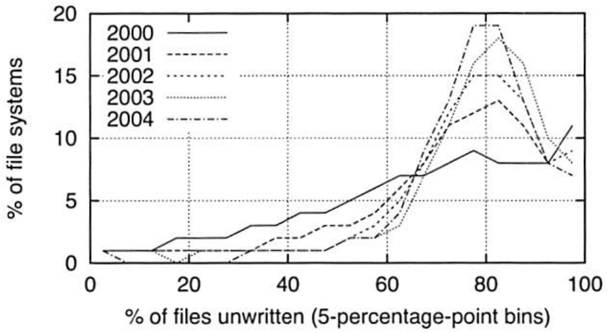


Figure 11: Histograms of file systems by percentage of files unwritten

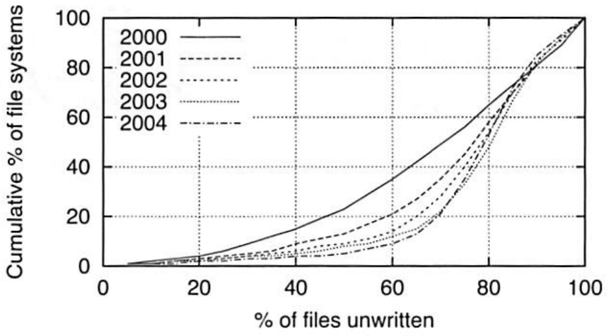


Figure 12: CDFs of file systems by percentage of files unwritten

virtual machine use is increasing. The null extension exhibits a notable anomaly in 2003, but we cannot investigate the cause without decrypting the file names in our datasets, which would violate our privacy policy.

Since files with the same extension have similar properties and requirements, some file system management policies can be improved by including special-case treatment for particular extensions. Such special-case treatment can be built into the file system or autonomically and dynamically learned [16]. Since nearly half the files, and nearly half the bytes, belong to files with a few popular extensions, developing such special-case treatment for only a few particular extensions can optimize performance for a large fraction of the file system. Furthermore, since the same extensions continue to be popular year after year, one can develop special-case treatments for today's popular extensions and expect that they will still be useful years from now.

3.5 Unwritten files

Figures 11 and 12 plot histograms and CDFs, respectively, of file systems by percentage of files that have not been written since they were copied onto the file system. We identify such files as ones whose modification timestamps are earlier than their creation timestamps, since the creation timestamp of a copied file is set to the time at which the copy was made, but its modification timestamp is copied from the original file. Over our sample period, the arithmetic mean of the percentage of locally unwritten files has grown from 66% to 76%, and the median has grown from 70% to 78%. This suggests that users locally contribute to a decreasing fraction of their systems' content. This may in part be due to the increasing amount of total content over time.

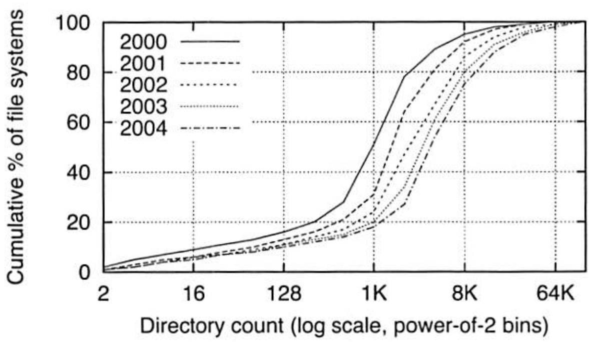


Figure 13: CDFs of file systems by directory count

Since more and more files are being copied across file systems rather than generated locally, we can expect identifying and coalescing identical copies to become increasingly important in systems that aggregate file systems. Examples of systems with such support are the FARSITE distributed file system [1], the Pastiche peer-to-peer backup system [8], and the Single Instance Store in Windows file servers [5].

4 Directories

4.1 Directory count per file system

Figure 13 plots CDFs of file systems by count of directories. The count of directories per file system has increased steadily over our five-year sample period: The arithmetic mean has grown from 2400 to 8900 directories and the median has grown from 1K to 4K directories.

We discussed implications of the rising number of directories per file system earlier, in §3.1.

4.2 Directory size

This section describes our findings regarding directory size, measured by count of contained files, count of contained subdirectories, and total entry count. None of these size distributions has changed appreciably over our sample period, but the mean count of files per directory has decreased slightly.

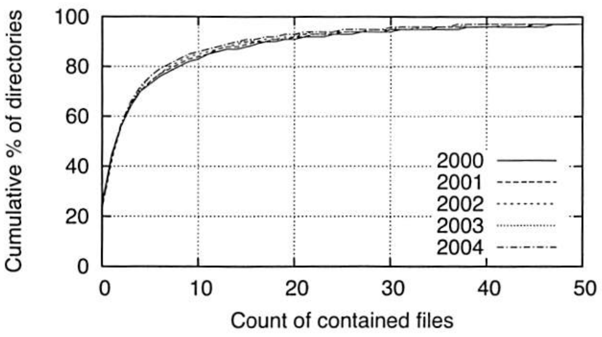


Figure 14: CDFs of directories by file count

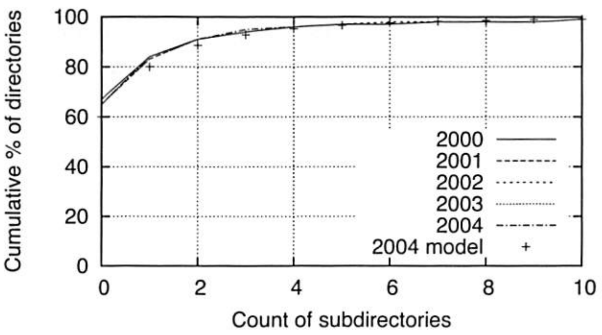


Figure 15: CDFs of directories by subdirectory count

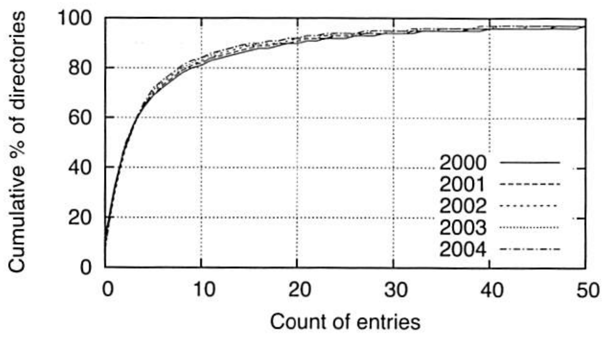


Figure 16: CDFs of directories by entry count

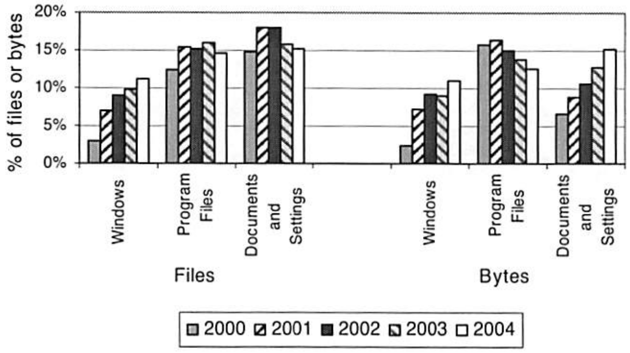


Figure 17: Fraction of files and bytes in special subtrees

Figure 14 plots CDFs of directories by size, as measured by count of files in the directory. It shows that although the absolute count of directories per file system has grown significantly over time, the distribution has not changed appreciably. Across all years, 23–25% of directories contain no files, which marks a change from 1998, in which only 18% contained no files and there were more directories containing one file than those containing none. The arithmetic mean directory size has decreased slightly and steadily from 12.5 to 10.2 over the sample period, but the median directory size has remained steady at 2 files.

Figure 15 plots CDFs of directories by size, as measured by count of subdirectories in the directory. It includes a model approximation we will discuss later in §4.5. This distribution has remained unchanged over our sample period. Across all years, 65–67% of directories contain no subdirectories, which is similar to the 69% found in 1998.

Figure 16 plots CDFs of directories by size, as measured by count of total entries in the directory. This distribution has remained largely unchanged over our sample period. Across all years, 46–49% of directories contain two or fewer entries.

Since there are so many directories with a small number of files, it would not take much space to colocate the metadata for most of those files with those directories. Such a layout would reduce seeks associated with file accesses. Therefore, it might be useful to preallocate a small amount of space near a new directory to hold a modest amount of child metadata. Similarly, most directories contain fewer than twenty entries, suggesting using an on-disk structure for directories that optimizes for this common case.

4.3 Special directories

This section describes our findings regarding the usage of Windows special directories. We find that an increasing fraction of file-system storage is in the namespace subtree devoted to system files, and the same holds for the subtree devoted to user documents and settings.

Figure 17 plots the fraction of file-system files that reside within subtrees rooted in each of three special directories: `Windows`, `Program Files`, and `Documents and Settings`. This figure also plots the fraction of file-system bytes contained within each of these special subtrees.

For the `Windows` subtree, the fractions of files and bytes have both risen from 2–3% to 11% over our sample period, suggesting that an increasingly large fraction of file-system storage is devoted to system files. In par-

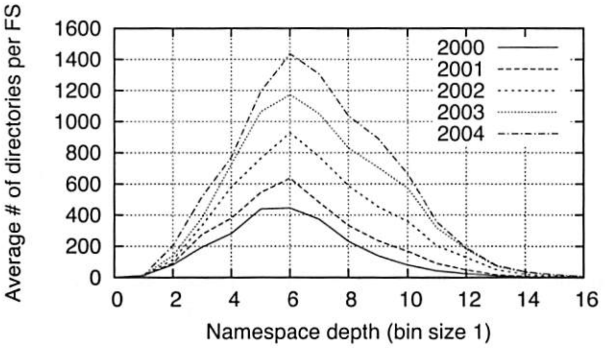


Figure 18: Histograms of directories by namespace depth

ticular, we note that Windows XP was released between the times of our 2000 and 2001 data collections.

For the `Program Files` subtree, the fractions of files and bytes have trended in opposite directions within the range of 12–16%. For the `Documents and Settings` subtree, the fraction of bytes has increased dramatically while the fraction of files has remained relatively stable.

The fraction of all files accounted for by these subtrees has risen from 25% to 40%, and the fraction of bytes therein has risen from 30% to 41%, suggesting that application writers and end users have increasingly adopted Windows' prescriptive namespace organization [7].

Backup software generally does not have to back up system files, since they are static and easily restored. Since system files are accounting for a larger and larger fraction of used space, it is becoming more and more useful for backup software to exclude these files.

On the other hand, files in the Documents and Settings folder tend to be the most important files to back up, since they contain user-generated content and configuration information. Since the percentage of bytes devoted to these files is increasing, backup capacity planners should expect, surprisingly, that their capacity requirements will increase *faster* than disk capacity is planned to grow. On the other hand, the percentage of files is not increasing, so they need not expect metadata storage requirements to scale faster than disk capacity. This may be relevant if metadata is backed up in a separate repository from the data, as done by systems such as EMC Centera [13].

4.4 Namespace tree depth

This section describes our findings regarding the depth of directories, files, and bytes in the namespace tree. We find that there are many files deep in the namespace tree, especially at depth 7. Also, we find that files deeper in the namespace tree tend to be orders-of-magnitude smaller than shallower files.

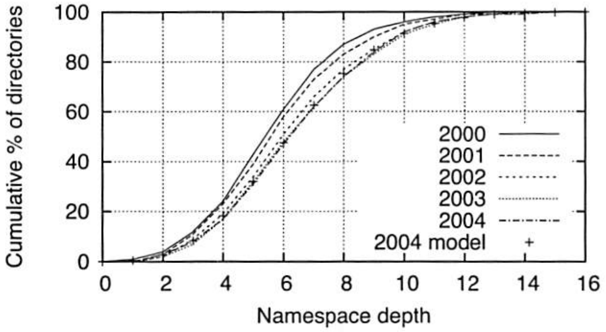


Figure 19: CDFs of directories by namespace depth

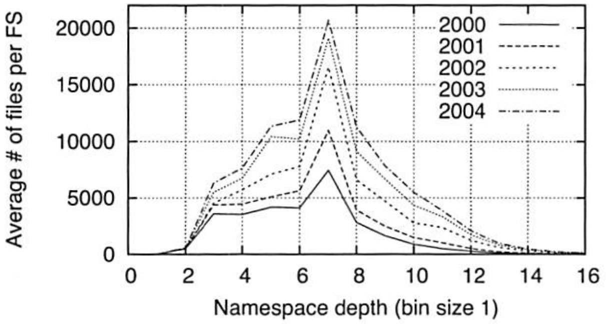


Figure 20: Histograms of files by namespace depth

Figure 18 plots histograms of directories by their depth in the namespace tree, and Figure 19 plots CDFs of this same data; it also includes a model approximation we will discuss later in §4.5. The general shape of the distribution has remained consistent over our sample period, but the arithmetic mean has grown from 6.1 to 6.9, and the median directory depth has increased from 5 to 6.

Figure 20 plots histograms of file count by depth in the namespace tree, and Figure 21 plots CDFs of this same data. With a few exceptions, such as at depths 2, 3, and 7, these distributions roughly track the observed distributions of directory depth, indicating that the count of files

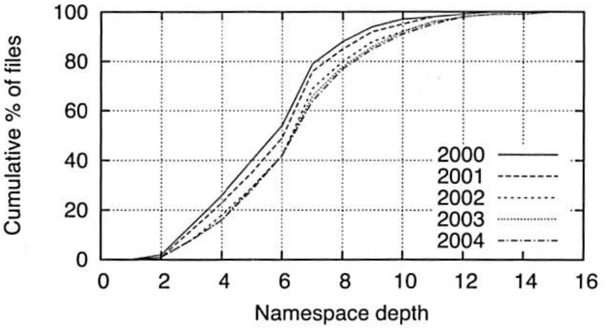


Figure 21: CDFs of files by namespace depth

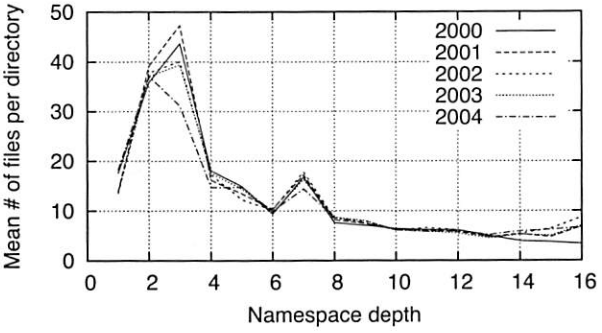


Figure 22: Files per directory vs. namespace depth

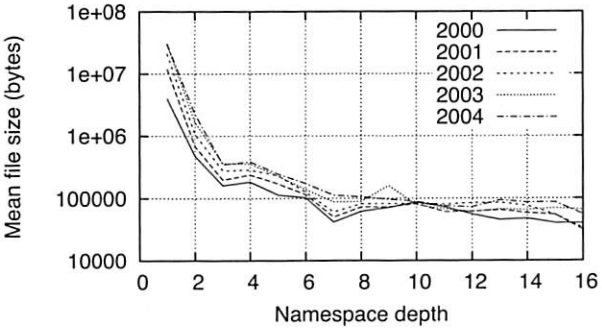


Figure 23: File size vs. namespace depth

per directory is mostly independent of directory depth. To study this more directly, Figure 22 plots the mean count of files per directory versus directory depth. There is a slight downward trend in this ratio with increasing depth, punctuated by three depths whose directories have greater-than-typical counts of files: At depth 2 are files in the `Windows` and `Program Files` directories; at depth 3 are files in the `System` and `System32` directories; and at depth 7 are files in the web cache directories.

Figure 23 plots the mean file size versus directory depth on a logarithmic scale. We see here that files deeper in the namespace tree tend to be smaller than shallower ones. The mean file size drops by two orders of magnitude between depth 1 and depth 3, and there is a drop of roughly 10% per depth level thereafter. This phenomenon occurs because most bytes are concentrated in a small number of large files (see Figures 2 and 4), and these files tend to reside in shallow levels of the namespace tree. In particular, the hibernation image file is located in the root.

Since many files and directories are deep in the namespace tree, efficient path lookup of deep paths should be a priority for file system designers. For instance, in distributed file systems where different servers are responsible for different parts of the namespace tree [1], deep path lookup may be expensive if not optimized. The high depth of many entries in the namespace may also be of interest to designers of file system visualization GUIs, to determine how much column space to allot for directory traversal. Furthermore, since the fraction of files at high depths is increasing across the years of our study, these lessons will become more and more important as years pass.

The clear trend of decreasing file size with increasing namespace tree depth sugests a simple coarse mechanism to predict future file size at time of file creation. File systems might use such prediction to decide where on disk to place a new file.

4.5 Namespace depth model

We have developed a generative model that accounts for the distribution of directory depth. The model posits that new subdirectories are created inside an existing directory in offset proportion to the count of subdirectories already in that directory.

In our previous study [9], we observed that the distribution of directories by depth could be approximated by a Poisson distribution with $\lambda = 4.38$, yielding a maximum displacement of cumulative curves (MDCC) of 2%. Poisson is also an acceptable approximation for the five datasets in the present study, with λ growing from 6.03 to 6.88 over the sample period, yielding MDCCs that range from 1% to 4%. However, the Poisson distribution does not provide an explanation for the behavior; it merely provides a means to approximate the result. By contrast, we have developed a generative model that accounts for the distribution of directory depths we have observed, with accuracy comparable to the Poisson model.

The generative model is as follows. A file system begins with an empty root directory. Directories are added to the file system one at a time. For each new directory, a parent directory is selected probabilistically, based on the count of subdirectories the parent currently has. Specifically, the probability of choosing each extant directory as a parent is proportional to $c(d) + 2$, where $c(d)$ is the count of extant subdirectories of directory d. We used Monte Carlo simulation to compute directory depth distributions according to this generative model. Given a count of directories in a file system, the model produces a distribution of directory depths that matches the observed distribution for file systems of that size. Figure 19 plots the aggregate result of the model for all file systems in the 2004 dataset. The model closely matches the CDF of observed directory depths, with an MDCC of 1%.

Our generative model accounts not only for the distribution of directory depth but also for that of subdirectory size. Figure 15 shows this for the 2004 dataset. The model closely matches the CDF, with an MDCC of 5%.

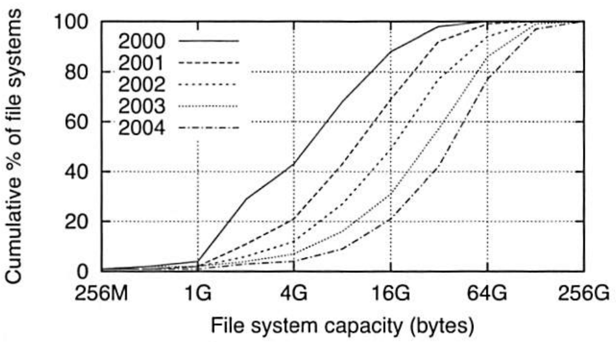


Figure 24: CDFs of file systems by storage capacity

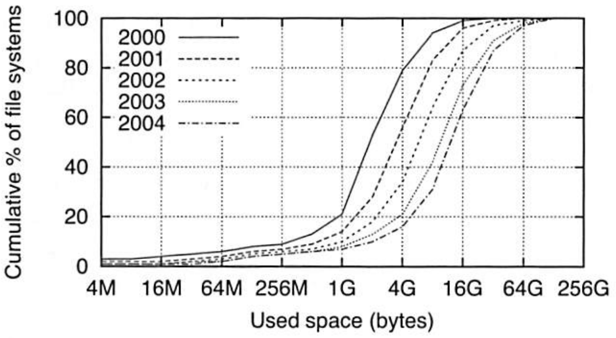


Figure 25: CDFs of file systems by total consumed space

Intuitively, the proportional probability $c(d) + 2$ can be interpreted as follows: If a directory already has some subdirectories, it has demonstrated that it is a useful location for subdirectories, and so it is a likely place for more subdirectories to be created. The more subdirectories it has, the more demonstrably useful it has been as a subdirectory home, so the more likely it is to continue to spawn new subdirectories. If the probability were proportional to $c(d)$ without any offset, then an empty directory could never become non-empty, so some offset is necessary. We found an offset of 2 to match our observed distributions very closely for all five years of our collected data, but we do not understand why the particular value of 2 should be appropriate.

5 Space Usage

5.1 Capacity and usage

Figure 24 plots CDFs of file system volumes by storage capacity, which has increased dramatically over our five-year sample period: The arithmetic mean has grown from 8 GB to 46 GB and the median has grown from 5 GB to 40 GB. The number of small-capacity file system volumes has dropped dramatically: Systems of 4 GB or less have gone from 43% to 4% of all file systems.

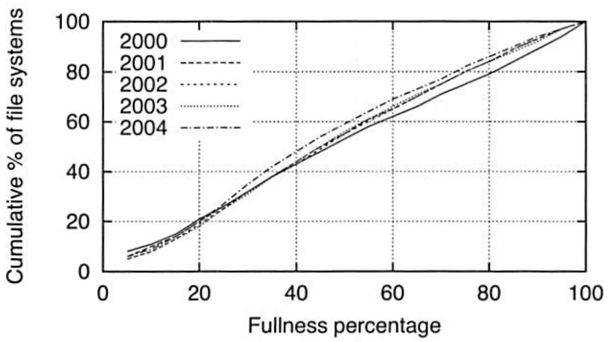


Figure 26: CDFs of file systems by fullness

Figure 25 plots CDFs of file systems by total consumed space, including not only file content but also space consumed by internal fragmentation, file metadata, and the system paging file. Space consumption increased steadily over our five-year sample period: The geometric mean has grown from 1 GB to 9 GB, the arithmetic mean has grown from 3 GB to 18 GB, and the median has grown from 2 GB to 13 GB.

Figure 26 plots CDFs of file systems by percentage of fullness, meaning the consumed space relative to capacity. The distribution is very nearly uniform for all years, as it was in our 1998 study. The mean fullness has dropped slightly from 49% to 45%, and the median file system has gone from 47% full to 42% full. By contrast, the aggregate fullness of our sample population, computed as total consumed space divided by total file-system capacity, has held steady at 41% over all years.

In any given year, the range of file system capacities in this organization is quite large. This means that software must be able to accommodate a wide range of capacities simultaneously existing within an organization. For instance, a peer-to-peer backup system must be aware that some machines will have drastically more capacity than others. File system designs, which must last many years, must accommodate even more dramatic capacity differentials.

5.2 Changes in usage

This subsection describes our findings regarding how individual file systems change in fullness over time. For this part of our work, we examined the 6536 snapshot pairs that correspond to the same file system in two consecutive years. We also examined the 1320 snapshot pairs that correspond to the same file system two years apart. We find that 80% of file systems become fuller over a one-year period, and the mean increase in fullness is 14 percentage points. This increase is predominantly due to creation of new files, partly offset by deletion of old files, rather than due to extant files changing size.

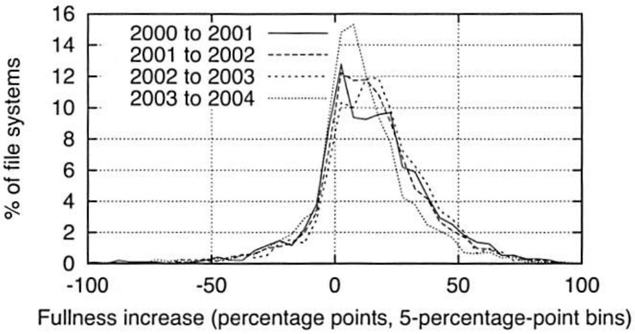


Figure 27: Histograms of file systems by 1-year fullness increase

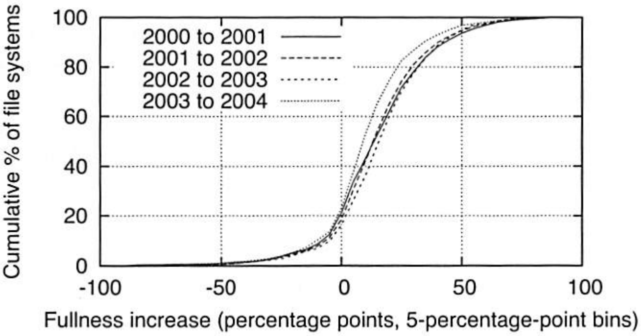


Figure 28: CDFs of file systems by 1-year fullness increase

When comparing two matching snapshots in different years, we must establish whether two files in successive snapshots of the same file system are the same file. We do not have access to files' inode numbers, because collecting them would have lengthened our scan times to an unacceptable degree. We thus instead use the following proxy for file sameness: If the files have the same full pathname, they are considered the same, otherwise they are not. This is a conservative approach: It will judge a file to be two distinct files if it or any ancestor directory has been renamed.

Figures 27 and 28 plot histograms and CDFs, respectively, of file systems by percentage-point increase in fullness from one year to the next. We define this term by example: If a file system was 50% full in 2000 and 60% full in 2001, it exhibited a 10 percentage-point increase in fullness. The distribution is substantially the same for all four pairs of consecutive years. Figure 28 shows that 80% of file systems exhibit an increase in fullness and fewer than 20% exhibit a decrease. The mean increase from one year to the next is 14 percentage points.

We also examined the increase in fullness over two years. We found the mean increase to be 22 percentage points. This is less than twice the consecutive-year increase, indicating that as file systems age, they increase their fullness at a slower rate. Because we have so few file systems with snapshots in four consecutive years, we did not explore increases over three or more years.

Since file systems that persist for a year tend to increase their fullness by about 14 points, but the mean file-system fullness has dropped from 49% to 45% over our sample period, it seems that the steadily increasing fullness of individual file systems is offset by the replacement of old file systems with newer, emptier ones.

Analyzing the factors that contribute to the 14-point mean year-to-year increase in fullness revealed the following breakdown: Fullness increases by 28 percentage points due to files that are present in the later snapshot but not in the earlier one, meaning that they were created during the intervening year. Fullness decreases by 15 percentage points due to files that are present in the earlier snapshot but not in the later one, meaning that they were deleted during the intervening year. Fullness also increases by 1 percentage point due to growth in the size of files that are present in both snapshots. An insignificant fraction of this increase is attributable to changes in system paging files, internal fragmentation, or metadata storage.

We examined the size distributions of files that were created and of files that were deleted, to see if they differed from the overall file-size distribution. We found that they do not differ appreciably. We had hypothesized that users tend to delete large files to make room for new content, but the evidence does not support this hypothesis.

Since deleted files and created files have similar size distributions, file system designers need not expect the fraction of files of different sizes to change as a file system ages. Thus, if they find it useful to assign different parts of the disk to files of different sizes, they can anticipate the allocation of sizes to disk areas to not need radical change as time passes.

Many peer-to-peer systems use free space on computers to store shared data, so the amount of used space is of great importance. With an understanding of how this free space decreases as a file system ages, a peer-to-peer system can proactively plan how much it will need to offload shared data from each file system to make room for additional local content. Also, since a common reason for upgrading a computer is because its disk space becomes exhausted, a peer-to-peer system can use a prediction of when a file system will become full as a coarse approximation to when that file system will become unavailable.

6 Related Work

This research extends our earlier work in measuring and modeling file-system metadata on Windows workstations. In 1998, we collected snapshots of over ten thousand file systems on the desktop computers at Microsoft [9]. The focus of the earlier study was on variations among file systems within the sample, all of which were captured at the same time. By contrast, the focus of the present study is on longitudinal analysis, meaning how file systems have changed over time.

Prior to our previous study, there were no studies of static file-system metadata on Windows systems, but there were several such studies in other operating-system environments. These include Satyanarayanan's study of a Digital PDP-10 at CMU in 1981 [24], Mullender and Tanenbaum's study of a Unix system at Vrije Universiteit in 1984 [20], Irlam's study of 1050 Unix file systems in 1993 [14], and Sienknecht et al.'s study of 267 file systems in 46 HP-UX systems at Hewlett-Packard in 1994 [25]. All of these studies involved snapshots taken at a single time, like our study in 1998. There have also been longitudinal studies of file-system metadata, but for significantly shorter times than ours: Bennett *et al.*studied three file servers at the University of Western Ontario over a period of one day in 1991 [4], and Smith and Seltzer studied 48 file systems on four file servers at Harvard over a period of ten months in 1994 [26].

We are aware of only one additional collection of static file-system metadata since our previous study. In 2001, Evans and Kuenning captured snapshots from 22 machines running various operating systems at Harvey Mudd College and Marine Biological Laboratories [11]. Their data collection and analysis focused mainly, but not exclusively, on media files. Their findings show that different types of files exhibit significantly different size distributions, which our results support.

Many studies have examined dynamic file-system traces rather than static file system snapshots. These studies are complementary to ours, describing things we cannot analyze such as the rate at which bytes are read and written in a file system. A few examples of such studies are Ousterhout *et al.*'s analysis of the BSD file system [21], Gribble *et al.*'s analysis of self-similarity in the dynamic behavior of various file systems [12], Vogels's analysis of Windows NT [28], and Roselli *et al.*'s analysis of HP-UX and Windows NT [23].

In addition to file-system measurement research, there has been much work in modeling file-system characteristics, most notably related to the distribution of file sizes. Examples of work in this area include that of Satyanarayanan [24], Barford and Crovella [3], Downey [10], and Mitzenmacher [19].

In 2001, Evans and Kuenning broke down measured file-size distributions according to file type, and they modeled the sizes using log-lambda distributions [11]. They found that video and audio files can significantly perturb the file-size distribution and prevent simple size models from applying. We did not find this to be true for file sizes in our sample population. However, we did find video, database, and blob files responsible for a second peak in the distribution of bytes by containing file size.

In our previous study, we modeled directory depth with a Poisson distribution [9], but we have herein proposed a generative model in which the attractiveness of an extant directory d as a location for a new subdirectory is proportional to $c(d) + 2$, where $c(d)$ is the count of directory d's extant subdirectories. This is strikingly similar to the rule for generating plane-oriented recursive trees, wherein the probability is proportional to $c(d) + 1$ [15].

7 Summary and Conclusions

Over a span of five years, we collected metadata snapshots from more than 63,000 distinct Windows file systems in a commercial environment, through voluntary participation of the systems' users. These systems contain 4 billion files totaling 700 TB of file data. For more than 10% of these file systems, we obtained snapshots in multiple years, enabling us to directly observe how these file systems have changed over time. Our measurements reveal several interesting properties of file systems and offer useful lessons.

One interesting discovery is the emergence of a second mode in the GB range in the distribution of bytes by containing file size. It makes us wonder if at some future time a third mode will arise. The increasingly large fraction of content in large files suggests that variable block sizes, as supported by ZFS [6] and NTFS [27], are becoming increasingly important. Since a few large files, mainly video, database, and blob files, are contributing to an increasing fraction of file-system usage, these file extensions are ideal candidates for larger block sizes.

Although large files account for a large fraction of space, most files are 4 KB or smaller. Thus, it is useful to colocate several small files in a single block, as ReiserFS [22] does, and to colocate small file content with file metadata, as NTFS does. Our finding that most directories have few entries suggests yet another possibility: Colocate small file content with the file's parent directory. An even more extreme solution is suggested by the fact that in 2004, the average file system had only 52 MB in files 4 KB or smaller. Since this number is becoming small relative to main memory sizes, it may soon be practical to avoid cache misses entirely for small files by prefetching them all at boot time and pinning them in the cache.

Another noteworthy discovery is that the fraction of files locally modified decreases with time, an effect significant enough to be observable in only a five-year sample. It would appear that users' ability to generate increasing amounts of content is outstripped by the phenomenal growth in their disks. If individuals copying content from each other becomes increasingly common, then applications like peer-to-peer backup will have increasing amounts of inter-machine content similarity to leverage to obviate copying.

We were surprised to find a strong negative correlation between namespace depth and file size. Such a strong and temporally-invariant correlation, in combination with the well-known correlation between file extension and file size, can help us make predictions of file size at creation time. This may be useful, e.g., to decide how many blocks to initially allocate to a file.

We also discovered that a simple generative model can account for both the distributions of directory depth and the count of subdirectories per directory. The model we developed posits that new subdirectories are created inside an existing directory in offset proportion to the count of subdirectories already in that directory. This behavior is easy to simulate, and it produces directory-depth and directory-size distributions that closely match our observations.

Finally, it is remarkable that file system fullness over the course of five years has changed little despite the vast increase in file system capacity over that same period. It seems clear that users scale their capacity needs to their available capacity. The lesson for storage manufacturers is to keep focusing effort on increasing capacity, because customers will continue to place great value on capacity for the foreseeable future.

8 Acknowledgments

The authors would like to express great thanks to the many Microsoft users who consented to us taking snapshots of their file systems. Without them, this study would have been impossible. We also thank Jim Gray, Catharine van Ingen, and Tim Harris for their comments on early drafts of this paper. Finally, we thank the anonymous reviewers and our shepherd, Randal Burns, for their many helpful comments and suggestions.

References

[1] ADYA, A., BOLOSKY, W., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J., HOWELL, J., LORCH, J., THEIMER, M., AND WATTENHOFER, R. P. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, December 2002), pp. 1–14.

[2] ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (Banff, Canada, October 2001), pp. 43–56.

[3] BARFORD, P., AND CROVELLA, M. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Madison, WI, June 1998), pp. 151–160.

[4] BENNETT, J. M., BAUER, M. A., AND KINCHLEA, D. Characteristics of files in NFS environments. In *Proceedings of the 1991 ACM SIGSMALL/PC Symposium on Small Systems* (Toronto, Ontario, June 1991), pp. 33–40.

[5] BOLOSKY, W. J., CORBIN, S., GOEBEL, D., AND DOUCEUR, J. R. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium* (Seattle, WA, August 2000).

[6] BONWICK, J. Zfs: The last word in file systems. Available at http://www.opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[7] CHAPMAN, G. Why does Explorer think I only want to see my documents? Available at http://pubs.logicalexpressions.com/Pub0009/LPMArticle.asp?ID=189.

[8] COX, L. P., MURRAY, C. D., AND NOBLE, B. D. Pastiche: Making backup cheap and easy. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, December 2002), pp. 285–298.

[9] DOUCEUR, J. R., AND BOLOSKY, W. J. A large-scale study of file-system contents. In *Proceedings of the 1999 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Atlanta, GA, May 1999), pp. 59–70.

[10] DOWNEY, A. B. The structural cause of file size distributions. In *Proceedings of the 2001 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Cambridge, MA, June 2001), pp. 328–329.

[11] EVANS, K. M., AND KUENNING, G. H. A study of irregularities in file-size distributions. In *Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)* (San Diego, CA, July 2002).

[12] GRIBBLE, S. D., MANKU, G. S., ROSELLI, D. S., BREWER, E. A., GIBSON, T. J., AND MILLER, E. L. Self-similarity in file systems. In *Proceedings of the 1998 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Madison, WI, June 1998), pp. 141–150.

[13] GUNAWI, H. S., AGRAWAL, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SCHINDLER, J. Deconstructing commodity storage clusters. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)* (Madison, WI, June 2005), pp. 60–71.

[14] IRLAM, G. Unix file size survey – 1993. Available at http://www.base.com/gordoni/ufs93.html.

[15] MAHMOUD, H. M. Distances in random plane-oriented recursive trees. *Journal of Computational and Applied Mathematics 41* (1992), 237–245.

[16] MESNIER, M., THERESKA, E., GANGER, G. R., ELLARD, D., AND SELTZER, M. File classification in self-* storage systems. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC)* (New York, NY, May 2004).

[17] MICROSOFT. SetFileTime. Available at MSDN, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50lrfsetfiletime.asp.

[18] MITCHELL, S. *Inside the Windows 95 file system*. O'Reilly and Associates, 1997.

[19] MITZENMACHER, M. Dynamic models for file sizes and double Pareto distributions. *Internet Mathematics 1*, 3 (2004), 305–333.

[20] MULLENDER, S. J., AND TANENBAUM, A. S. Immediate files. *Software—Practice and Experience 14*, 4 (April 1984), 365–368.

[21] OUSTERHOUT, J. K., COSTA, H. D., HARRISON, D., KUNZE, J. A., KUPFER, M., AND THOMPSON, J. G. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP)* (Orcas Island, WA, December 1985), pp. 15–24.

[22] REISER, H. Three reasons why ReiserFS is great for you. Available at http://www.namesys.com/.

[23] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference* (San Diego, CA, June 2000), pp. 41–54.

[24] SATYANARAYANAN, M. A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)* (Pacific Grove, CA, December 1981), pp. 96–108.

[25] SIENKNECHT, T. F., FRIEDRICH, R. J., MARTINKA, J. J., AND FRIEDENBACH, P. M. The implications of distributed data in a commercial environment on the design of hierarchical storage management. *Performance Evaluation 20*, 1–3 (May 1994), 3–25.

[26] SMITH, K., AND SELTZER, M. File layout and file system performance. Technical Report TR-35-94, Harvard University, 1994.

[27] SOLOMON, D. A. *Inside Windows NT*, 2nd ed. Microsoft Press, 1998.

[28] VOGELS, W. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)* (Kiawah Island, SC, December 1999), pp. 93–109.

Proportional-Share Scheduling for Distributed Storage Systems

Yin Wang*
University of Michigan
yinw@eecs.umich.edu

Arif Merchant
HP Laboratories
arif@hpl.hp.com

Abstract

Fully distributed storage systems have gained popularity in the past few years because of their ability to use cheap commodity hardware and their high scalability. While there are a number of algorithms for providing differentiated quality of service to clients of a centralized storage system, the problem has not been solved for distributed storage systems. Providing performance guarantees in distributed storage systems is more complex because clients may have different data layouts and access their data through different coordinators (access nodes), yet the performance guarantees required are global.

This paper presents a distributed scheduling framework. It is an adaptation of fair queuing algorithms for distributed servers. Specifically, upon scheduling each request, it enforces an extra delay (possibly zero) that corresponds to the amount of service the client gets on other servers. Different performance goals, e.g., per storage node proportional sharing, total service proportional sharing or mixed, can be met by different delay functions. The delay functions can be calculated at coordinators locally so excess communication is avoided. The analysis and experimental results show that the framework can enforce performance goals under different data layouts and workloads.

1 Introduction

The storage requirements of commercial and institutional organizations are growing rapidly. A popular approach for reducing the resulting cost and complexity of management is to consolidate the separate computing and storage resources of various applications into a common pool. The common resources can then be managed together and shared more efficiently. Distributed storage systems, such as *Federated Array of Bricks* (FAB) [20], Petal [16], and IceCube [27], are designed to serve as large storage pools. They are built from a number of individual storage nodes, or bricks, but present a single, highly-available store to users. High scalability is another advantage of distributed storage systems. The system can grow smoothly from small to large-scale installations because it is not limited by the capacity of an array or mainframe chassis. This satisfies the needs of service providers to continuously add application workloads onto storage resources.

A data center serving a large enterprise may support thousands of applications. Inevitably, some of these applications will have higher storage performance requirements than others. Traditionally, these requirements have been met by allocating separate storage for such applications; for example, applications with high write rates may be allocated storage on high-end disk arrays with large caches, while other applications live on less expensive, lower-end storage. However, maintaining separate storage hardware in a data center can be a management nightmare. It would be preferable to provide each application with the service level it requires while sharing storage. However, storage systems typically treat all I/O requests equally, which makes differentiated service difficult. Additionally, a bursty I/O workload from one application can cause other applications sharing the same storage to suffer.

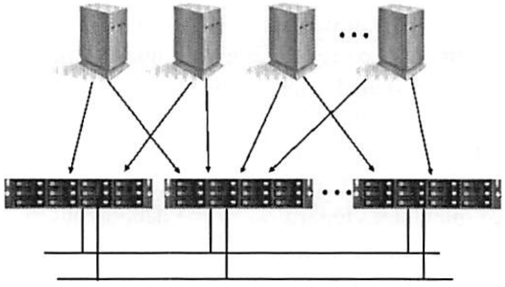


Figure 1: **A distributed storage system**

One solution to this problem is to specify the performance requirement of each application's storage workload and enable the storage system to ensure that it is met. Thus applications are insulated from the impact of workload surges in other applications. This can be achieved by ordering the requests from the applications appropriately, usually through a centralized scheduler, to coordinate access to the shared resources [5, 6, 24]. The scheduler can be implemented in the server or as a separate *interposed request scheduler* [2, 12, 17, 29] that treats the storage server as a black box and applies the resource control externally.

Centralized scheduling methods, however, fit poorly with distributed storage systems. To see this, consider the typical distributed storage system shown in Figure 1. The system is composed of bricks; each brick is a computer with a CPU, memory, networking, and storage. In a symmetric system, each brick runs the same software. Data stored by the system is distributed across the bricks. Typically, a client accesses the data through a *coordina-*

*This work was done during an internship at HP Laboratories.

tor, which locates the bricks where the data resides and performs the I/O operation. A brick may act both as a storage node and a coordinator. Different requests, even from the same client, may be coordinated by different bricks. Two features in this distributed architecture prevent us from applying any existing request scheduling algorithm directly. First, the coordinators are distributed. A coordinator schedules requests possibly without the knowledge of requests processed by other coordinators. Second, the data corresponding to requests from a client could be distributed over many bricks, since a logical volume in a distributed storage system may be striped, replicated, or erasure-coded across many bricks [7]. Our goal is to design a distributed scheduler that can provide service guarantees regardless of the data layout.

This paper proposes a distributed algorithm to enforce *proportional sharing* of storage resources among *streams* of requests. Each stream has an assigned *weight*, and the algorithm reserves for it a minimum share of the system capacity proportional to its weight. Surplus resources are shared among streams with outstanding requests, also in proportion to their weights. System capacity, in this context, can be defined in a variety of ways: for example, the number of I/Os per second, the number of bytes read or written per second, etc. The algorithm is work-conserving: no resource is left idle if there is any request waiting for it. However, it can be shown easily that a work-conserving scheduling algorithm for multiple resources (bricks in our system) cannot achieve proportional sharing in all cases. We present an extension to the basic algorithm that allows per-brick proportional sharing in such cases, or a method that provides a hybrid between system-wide proportional sharing and per-brick proportional sharing. This method allows total proportional sharing when possible while ensuring a minimum level of service on each brick for all streams.

The contribution of this paper includes a novel distributed scheduling framework that can incorporate many existing centralized fair queuing algorithms. Within the framework, several algorithms that are extensions to Start-time Fair Queuing [8] are developed for different system settings and performance goals. To the best of our knowledge, this is the first algorithm that can achieve total service proportional sharing for distributed storage resources with distributed schedulers. We evaluate the proposed algorithms both analytically and experimentally on a FAB system, but the results are applicable to most distributed storage systems. The results confirm that the algorithms allocate resources fairly under various settings — different data layouts, clients accessing the data through multiple coordinators, and fluctuating service demands.

This paper is organized as follows. Section 2 presents an overview of the problem, the background, and the related work. Section 3 describes our distributed fair queueing framework, two instantiations of it, and their properties. Section 4 presents the experimental evaluation of the algorithms. Section 5 concludes.

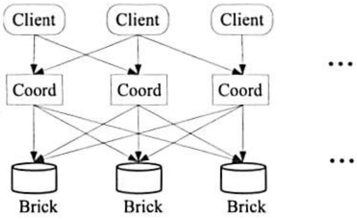


Figure 2: **Data access model of a distributed storage system.** Different clients may have different data layouts spreading across different sets of bricks. However, coordinators know all data layouts and can handle requests from any client.

2 Overview and background

We describe here the distributed storage system that our framework is designed for, the proportional sharing properties it is intended to enforce, the centralized algorithm that we base our work upon, and other related work.

2.1 Distributed Storage Systems

Figure 2 shows the configuration of a typical distributed storage system. The system includes a collection of storage *bricks*, which might be built from commodity disks, CPUs, and NVRAM. Bricks are connected by a standard network such as gigabit Ethernet. Access to the data on the bricks is handled by the coordinators, which present a *virtual disk* or *logical volume* interface to the clients. In the FAB distributed storage system [20], a client may access data through an arbitrary coordinator or a set of coordinators at the same time to balance its load. Coordinators also handle data layout and volume management tasks, such as volume creation, deletion, extension and migration. In FAB, the coordinators reside on the bricks, but this is not required. We consider local area distributed storage systems where the network latencies are small compared with disk latencies. We assume that the network bandwidths are sufficiently large that the I/O throughput is limited by the bricks rather than the network.

The data layout is usually designed to optimize properties such as load balance, availability, and reliability. In FAB, a logical volume is divided into a number of *segments*, which may be distributed across bricks using a replicated or erasure-coded layout. The choice of brick-set for each segment is determined by the storage system. Generally, the layout is opaque to the clients.

The scheduling algorithm we present is designed for such a distributed system, making a minimum of as-

SYMBOLS	DESCRIPTION
ϕ_f	Weight of stream f
p^i_f	Stream f's i-th request
$p^i_{f,A}$	Stream f's i-th request to brick A
$cost(\cdot)$	Cost of a single request
$cost^{max}_f$	Max request cost of stream f
$cost^{max}_{f,A}$	Max cost on brick A of f
$W_f(t_1, t_2)$	Aggregate cost of requests served from f during interval $[t_1, t_2]$
$batchcost$ $(p^i_{f,A})$	Total cost of requests in between $p^{i-1}_{f,A}$ and $p^i_{f,A}$, including $p^i_{f,A}$
$batchcost^{max}_{f,A}$	Max value of $batchcost(p^i_{f,A})$
$A(\cdot)$	Arrival time of a request
$S(\cdot)$	Start tag of a request
$F(\cdot)$	Finish tag of a request
$v(t)$	Virtual time at time t
$delay(\cdot)$	Delay value of a request

Table 1: **Some symbols used in this paper.**

sumptions. The data for a client may be laid out in an arbitrary manner. Clients may request data located on an arbitrary set of bricks at arbitrary and even fluctuating rates, possibly through an arbitrary set of coordinators.

2.2 Proportional Sharing

The algorithms in this paper support proportional sharing of resources for clients with queued requests. Each client is assigned a weight by the user and, in every time interval, the algorithms try to ensure that clients with requests pending during that interval receive service proportional to their weights.

More precisely, I/O requests are grouped into service classes called *streams*, each with a weight assigned; e.g., all requests from a client could form a single stream. A stream is *backlogged* if it has requests queued. A stream f consists a sequence of requests $p^0_f ... p^n_f$. Each request has an associated service cost $cost(p^i_f)$. For example, with bandwidth performance goals, the cost might be the size of the requested data; with service time goals, request processing time might be the cost. The maximum request cost of stream f is denoted $cost^{max}_f$. The weight assigned to stream f is denoted ϕ_f; only the relative values of the weights matter for proportional sharing.

Formally, if $W_f(t_1, t_2)$ is the aggregate cost of the requests from stream f served in the time interval $[t_1, t_2]$, then the unfairness between two continuously backlogged streams f and g is defined to be:

$$\left| \frac{W_f(t_1, t_2)}{\phi_f} - \frac{W_g(t_1, t_2)}{\phi_g} \right| \qquad (1)$$

A fair proportional sharing algorithm should guarantee that (1) is bounded by a constant. The constant usually depends on the stream characteristics, e.g., $cost^{max}_f$. The time interval $[t_1, t_2]$ in (1) may be any time duration. This corresponds to a "use it or lose it" policy, i.e., a stream will not suffer during one time interval for consuming surplus resources in another interval, nor will it benefit later from under-utilizing resources.

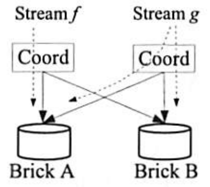


Figure 3: **Distributed data.** Stream f sends requests to brick A only while stream g sends requests to both A and B.

In the case of distributed data storage, we need to define what is to be proportionally shared. Let us first look at the following example.

EXAMPLE 1. Figure 3 is a storage system consisting of two bricks A and B. If streams f and g are equally weighted and both backlogged at A, how should we allocate the service capacity of brick A? ■

There are two alternatives for the above example, which induce two different meanings for proportional sharing. The first is *single brick proportional sharing*, i.e., service capacity of brick A will be proportionally shared. Many existing proportional sharing algorithms fall into this category. However, stream g also receives service at brick B, thus receiving higher overall service. While this appears fair because stream g does a better job of balancing its load over the bricks than stream f, note that the data layout may be managed by the storage system and opaque to the clients; thus the quality of load balancing is merely an accident. From the clients' point of view, stream f unfairly receives less service than stream g. The other alternative is *total service proportional sharing*. In this case, the share of the service stream f receives on brick A can be increased to compensate for the fact that stream g receives service on brick B, while f does not. This problem is more intricate and little work has been done on it.

It is not always possible to guarantee total service proportional sharing with a work-conserving scheduler, i.e., where the server is never left idle when there is a request queued. Consider the following extreme case.

EXAMPLE 2. Stream f requests service from brick A only, while equally weighted stream g is sending requests to A and many other bricks. The amount of service g obtains from the other bricks is larger than the capacity of A. With a work-conserving scheduler, it is impossible to equalize the total service of the two streams. ■

If the scheduler tries to make the total service received by f and g as close to equal as possible, g will be blocked at brick A, which may not be desirable. Inspired by the example, we would like to guarantee some minimum service on each brick for each stream, yet satisfy total service proportional sharing whenever possible.

In this paper, we propose a distributed algorithm framework under which single brick proportional sharing, total service proportional sharing, and total service proportional sharing with a minimum service guarantee are all possible. We note that, although we focus on distributed storage systems, the algorithms we propose may be more broadly applicable to other distributed systems.

2.3 The centralized approach

In selecting an approach towards a distributed proportional sharing scheduler, we must take four requirements into account: i) the scheduler must be work-conserving: resources that backlogged streams are waiting for should never be idle; ii) "use it or lose it", as described in the previous section; iii) the scheduler should accommodate fluctuating service capacity since the service times of IOs can vary unpredictably due to the effects of caching, sequentiality, and interference by other streams; and iv) reasonable computational complexity—there might be thousands of bricks and clients in a distributed storage system, hence the computational and communication costs must be considered.

There are many centralized scheduling algorithms that could be extended to distributed systems [3, 4, 8, 28, 30, 17, 14, 9]. We chose to focus on the Start-time Fair Queuing (SFQ) [8] and its extension SFQ(D) [12] because they come closest to meeting the requirements above. We present a brief discussion of the SFQ and SFQ(D) algorithms in the remainder of this section.

SFQ is a proportional sharing scheduler for a single server; intuitively, it works as follows. SFQ assigns a *start time* tag and a *finish time* tag to each request corresponding to the normalized times at which the request should start and complete according to a system notion of *virtual time*. For each stream, a new request is assigned a start time based on the assigned finish time of the previous request, or the current virtual time, whichever is greater. The finish time is assigned as the start time plus the normalized cost of the request. The virtual time is set to be the start time of the currently executing request, or the finish time of the last completed request if there is none currently executing. Requests are scheduled in the order of their start tags. It can be shown that, in any time interval, the service received by two backlogged workloads is approximately proportionate to their weights.

More formally, the request p_f^i is assigned the *start tag* $S(p_f^i)$ and the *finish tag* $F(p_f^i)$ as follows:

$$S(p_f^i) = \max\{v(A(p_f^i)), F(p_f^{i-1})\}, i \geq 1 \quad (2)$$

$$F(p_f^i) = S(p_f^i) + \frac{cost(p_f^i)}{\phi_f}, i \geq 1 \quad (3)$$

where $A(p_f^i)$ is the arrival time of request p_f^i, and $v(t)$ is the virtual time at t; $F(p_f^0) = 0, v(0) = 0$.

SFQ cannot be directly applied to storage systems, since storage servers are concurrent, serving multiple requests at a time, and the virtual time $v(t)$ is not defined. Jin *et al.* [12] extended SFQ to concurrent servers by defining the virtual time as the maximum start tag of requests in service (the last request dispatched). The resulting algorithm is called *depth-controlled start-time fair queuing* and abbreviated to SFQ(D), where D is the queue depth of the storage device. As with SFQ, the following theorem [12] shows that SFQ(D) provides backlogged workloads with proportionate service, albeit with a looser bound on the unfairness.

THEOREM 1. *During any interval* $[t_1, t_2]$, *the difference between the amount of work completed by an SFQ(D) server for two backlogged streams f and g is bounded by:*

$$\left| \frac{W_f(t_1,t_2)}{\phi_f} - \frac{W_g(t_1,t_2)}{\phi_g} \right| \leq \left(\frac{cost_f^{max}}{\phi_f} + \frac{cost_g^{max}}{\phi_g} \right) * (D+1) \quad (4)$$

While there are more complex variations of SFQ [12] that can reduce the unfairness of SFQ(D), for simplicity, we use SFQ(D) as the basis for our distributed scheduling algorithms. Since the original SFQ algorithm cannot be directly applied to storage systems, for the sake of readability, we will use "SFQ" to refer to SFQ(D) in the remainder of the paper.

2.4 Related Work

Extensive research in scheduling for packet switching networks has yielded a series of fair queuing algorithms; see [19, 28, 8, 3]. These algorithms have been adapted to storage systems for service proportional sharing. For example, YFQ [1], SFQ(D) and FSFQ(D) [12] are based on start-time fair queueing [8]; SLEDS [2] and SARC [29] use leaky buckets; CVC [11] employs the virtual clock [30]. Fair queuing algorithms are popular for two reasons: 1) they provide theoretically proven strong fairness, even under fluctuating service capacity, and 2) they are work-conserving.

However, fair queuing algorithms are not convenient for real-time performance goals, such as latencies. To address this issue, one approach is based on real-time schedulers; e.g., Façade [17] implements an Earliest

Deadline First (EDF) queue with the proportional feedback for adjusting the disk queue length. Another method is feedback control, a classical engineering technique that has recently been applied to many computing systems [10]. These generally require at least a rudimentary model of the system being controlled. In the case of storage systems, whose performance is notoriously difficult to model [22, 25], Triage [14] adopts an adaptive controller that can automatically adjust the system model based on input-output observations.

There are some frameworks [11, 29] combining the above two objectives (proportional sharing and latency guarantees) in a two-level architecture. Usually, the first level guarantees proportional sharing by fair queueing methods, such as CVC [11] and SARC [29]. The second level tries to meet the latency goal with a real-time scheduler, such as EDF. Some feedback from the second level to the first level scheduler is helpful to balance the two objectives [29]. All of the above methods are designed for use in a centralized scheduler and cannot be directly applied to our distributed scheduling problem.

Existing methods for providing quality of service in distributed systems can be put into two categories. The first category is the distributed scheduling of a single resource. The main problem here is to maintain information at each scheduler regarding the amount of resource each stream has so far received. For example, in fair queuing algorithms, where there is usually a system virtual time $v(t)$ representing the normalized fair amount of service that all backlogged clients should have received by time t, the problem is how to synchronize the virtual time among all distributed schedulers. This can be solved in a number or ways; for example, in high capacity crossbar switches, in order to fairly allocate the bandwidth of the output link, the virtual time of different input ports can be synchronized by the *access buffer* inside the crossbar [21]. In wireless networks, the communication medium is shared. When a node can overhear packages from neighboring nodes for synchronization, distributed priority backoff schemes closely approximate a single global fair queue [18, 13, 23]. In the context of storage scheduling, Request Window [12] is a distributed scheduler that is similar to a leaky bucket scheduler. Services for different clients are balanced by the windows issued by the storage server. It is not fully work-conserving under light workloads.

The second category is centralized scheduling of multiple resources. Gulati and Varman [9] address the problem of allocating disk bandwidth fairly among concurrent competing flows in a parallel I/O system with multiple disks and a centralized scheduler. They aim at the optimization problem of minimizing the unfairness among different clients with concurrent requests. I/O requests are scheduled in batches, and a combinatorial optimization problem is solved in each round, which makes the method computationally expensive. The centralized controller makes it unsuitable for use in fully distributed high-performance systems, such as FAB.

To the best of our knowledge, the problem of fair scheduling in distributed storage systems that involve both distributed schedulers and distributed data has not been previously addressed.

3 Proportional Sharing in Distributed Storage Systems

We describe a framework for proportional sharing in distributed storage systems, beginning with the intuition, followed by a detailed description, and two instantiations of the method exhibiting different sharing properties.

3.1 An intuitive explanation

First, let us consider the simplified problem where the data is centralized at one brick, but the coordinators may be distributed. An SFQ scheduler could be placed either at coordinators or at the storage brick. As fair scheduling requires the information for all backlogged streams, direct or indirect communication among coordinators may be necessary if the scheduler is implemented at coordinators. Placing the scheduler at bricks avoids the problem. In fact, SFQ(D) can be used without modification in this case, provided that coordinators attach a stream ID to each request so that the scheduler at the brick can assign the start tag accordingly.

Now consider the case where the data is distributed over multiple bricks as well. In this case, SFQ schedulers at each brick can guarantee only single brick proportional sharing, but not necessarily total service proportional sharing because the scheduler at each brick sees only the requests directed to it and cannot account for the service rendered at other bricks.

Suppose, however, that each coordinator broadcasts all requests to all bricks. Clearly, in this case, each brick has complete knowledge of all requests for each stream. Each brick responds only to the requests for which it is the correct destination. The remaining requests are treated as *virtual requests*, and we call the combined stream of real and virtual request a *virtual stream*; see Fig. 4. A virtual request takes zero processing time but does account for the service share allocated to its source stream. Then the SFQ scheduler at the brick guarantees service proportional sharing of backlogged virtual streams. As the aggregate service cost of a virtual stream equals the aggregate service cost of the original stream, total service proportional sharing can be achieved.

The above approach is simple and straightforward, but with large-scale distributed storage systems, broadcast-

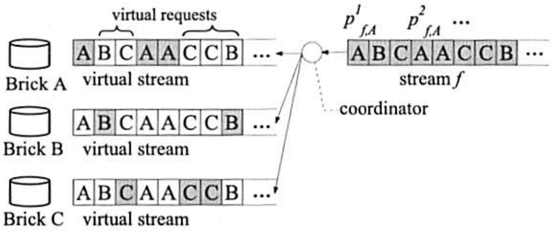


Figure 4: **The naive approach.** The coordinator broadcasts every request to all bricks. Requests to incorrect destination bricks are *virtual* and take zero processing time. Proportional scheduling at each local brick guarantees total service proportional sharing.

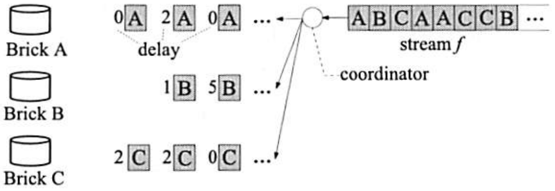


Figure 5: **The improved approach.** Only the aggregate cost of virtual requests is communicated, indicated by the number before each request (assuming unit cost of each request). Broadcasting is avoided yet total service proportional sharing can be achieved.

ing is not acceptable. We observe, however, that the SFQ scheduler requires only knowledge of the cost of each virtual request, the coordinators may therefore broadcast the cost value instead of the request itself. In addition, the coordinator may combine the cost of consecutive virtual requests and piggyback the total cost information onto the next real request; see Fig. 5. The communication overhead is negligible because, in general, read/write data rather than requests dominate the communication and the local area network connecting bricks usually has enough bandwidth for this small overhead.

The piggyback cost information on each real request is called the *delay* of the request, because the modified SFQ scheduler will delay processing the request according to this value. Different delay values may be used for different performance goals, which greatly extends the ability of SFQ schedulers. This flexibility is captured in the framework presented next.

3.2 Distributed Fair Queuing Framework

We propose the distributed fair queuing framework displayed in Fig. 6; as we show later, it can be used for total proportional sharing, single-brick proportional sharing, or a hybrid between the two. Assume there are streams $f, g, ...$ and bricks $A, B, ...$. The fair queueing scheduler is placed at each brick as just discussed. The scheduler has a priority queue for all streams and orders all requests by some priority, e.g., start time tags in the case of an SFQ scheduler. On the other hand, each coordinator has a separate queue for each stream, where the requests in a queue may have different destinations.

When we apply SFQ to the framework, each request has a start tag and a finish tag. To incorporate the idea presented in the previous section, we modify the computation of the tags as follows:

$$S(p^i_{f,A}) = \max \left\{ v(A(p^i_{f,A})), F(p^{i-1}_{f,A}) + \frac{delay(p^i_{f,A})}{\phi_f} \right\} \quad (5)$$

$$F(p^i_{f,A}) = S(p^i_{f,A}) + \frac{cost(p^i_{f,A})}{\phi_f} \quad (6)$$

The only difference between SFQ formulae (2-3) and those above is the new delay function for each request, which is calculated at coordinators and carried by the request. The normalized delay value translates into the amount of time by which the start tag should be shifted. How the delay is computed depends upon the proportional sharing properties we wish to achieve, and we will discuss several delay functions and the resulting sharing properties in the sections that follow. We will refer to the modified Start-time Fair Queuing algorithm as Distributed Start-time Fair Queuing (DSFQ).

In DSFQ, as in SFQ(D), $v(t)$ is defined to be the start tag of the last request dispatched to the disk before or at time t. There is no global virtual time in the system. Each brick maintains its own virtual time, which varies at different bricks depending on the workload and the service capacity of the brick.

We note that the framework we propose works with other fair scheduling algorithms [28] as long as each stream has its own clock such that the delay can be applied; for example, a similar extension could be made to the Virtual Clock algorithm [30] if we desire proportional service over an extended time period (time-averaged fairness) rather than the "use it or lose it" property (instantaneous fairness) supported by SFQ. Other options between these two extreme cases could be implemented in this framework, as well. For example, the scheduler can constrain each stream's time tag to be within some window of the global virtual time. Thus, a stream that underutilizes its share can get extra service later, but only to a limited extent.

If the delay value is set to always be zero, DSFQ reduces to SFQ and achieves single brick proportional sharing. We next consider other performance goals.

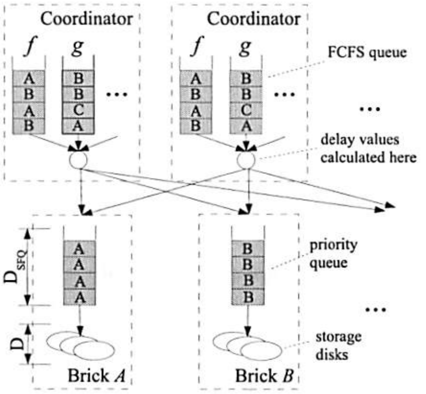


Figure 6: **The distributed fair queuing framework**

3.3 Total Service Proportional Sharing

We describe how the distributed fair queueing framework can be used for total proportional sharing when each stream uses one coordinator, and then argue that the same method also engenders total proportional sharing with multiple coordinators per stream.

3.3.1 Single-Client Single-Coordinator

We first assume that requests from one stream are always processed by one coordinator; different streams may or may not have different coordinators. We will later extend this to the multiple coordinator case. The performance goal, as before, is that the total amount of service each client receives must be proportional to its weight.

As described in Section 3.1, the following delay function for a request from stream f to brick A represents the total cost of requests sent to other bricks since the previous request to brick A.

$$delay(p^i_{f,A}) \;=\; batchcost(p^i_{f,A}) - cost(p^i_{f,A}) \quad (7)$$

When this delay function is used with the distributed scheduling framework defined by formulae (5-7), we call the resulting algorithm TOTAL-DSFQ. The delay function (7) is the total service cost of requests sent to other bricks since the last request on the brick. Intuitively, it implies that, if the brick is otherwise busy, a request should wait an extra time corresponding to the aggregate service requirements of the preceding requests from the same stream that were sent to other bricks, normalized by the stream's weight.

Why TOTAL-DSFQ engenders proportional sharing of the total service received by the streams can be explained using virtual streams. According to the formulae (5-7), TOTAL-DSFQ is exactly equivalent to the architecture where coordinators send virtual streams to the bricks and bricks are controlled by the standard SFQ. This virtual stream contains all the requests in f, but the requests that are not destined for A are served at A in zero time. Note that SFQ holds its fairness property even when the service capacity varies [8]. In our case, the server capacity (processing speed) varies from normal, if the request is to be serviced on the same brick, to infinity if the request is virtual and is to be serviced elsewhere. Intuitively, since the brick A sees all the requests in f (and their costs) as a part of the virtual stream, the SFQ scheduler at A factors in the costs of the virtual requests served elsewhere in its scheduling, even though they consume no service time at A. This will lead to proportional sharing of the total service. The theorem below formalizes the bounds on unfairness using TOTAL-DSFQ.

THEOREM 2. *Assume stream f is requesting service on N_f bricks and stream g on N_g bricks. During any interval $[t_1, t_2]$ in which f and g are both continuously backlogged at some brick A, the difference between the total amount of work completed by all bricks for the two streams during the entire interval, normalized by their weights, is bounded as follows:*

$$\begin{aligned}
&\left|\frac{W_f(t_1,t_2)}{\phi_f} - \frac{W_g(t_1,t_2)}{\phi_g}\right| \\
&\leq ((D + D_{SFQ}) * N_f + 1)\frac{cost^{max}_{f,A}}{\phi_f} + \\
&((D + D_{SFQ}) * N_g + 1)\frac{cost^{max}_{g,A}}{\phi_g} + \\
&(D_{SFQ} + 1)\left(\frac{batchcost^{max}_{f,A}}{\phi_f} + \frac{batchcost^{max}_{g,A}}{\phi_g}\right) \quad (8)
\end{aligned}$$

where D is the queue depth of the disk[1]*, and D_{SFQ} is the queue depth of the Start-time Fair Queue at the brick.*

PROOF. The proof of this and all following theorems can be found in [26]. □

The bound in Formula (8) has two parts. The first part is similar to the bound of SFQ(D) in (4), the unfairness due to server queues. The second part is new and contributed by the distributed data. If the majority of requests of stream f is processed at the backlogged server, the $batchcost^{max}_{f,A}$ is small and the bound is tight. Otherwise, if f gets a lot of service at other bricks, the bound is loose.

As we showed in Example 2, however, there are situations in which total proportional sharing is impossible with work conserving schedulers. In the theorem above, this corresponds to the case with an infinite $batchcost^{max}_{g,A}$, and hence the bound is infinite. To delineate more precisely when total proportional sharing is possible under TOTAL-DSFQ, we characterize when the total service rates of the streams are proportional to their weights. The theorem below says that, under TOTAL-DSFQ, if a set of streams are backlogged together at a

[1] If there are multiple disks (the normal case), D is the sum of the queue depths of the disks.

set of bricks, then either their normalized total service rates over all bricks are equal (thus satisfying the total proportionality requirement), or there are some streams whose normalized service rates are equal and the remainder receive no service at the backlogged bricks because they already receive more service elsewhere.

Let $R_f(t_1, t_2) = W_f(t_1, t_2)/(\phi_f * (t_2 - t_1))$ be the normalized service rate of stream f in the duration (t_1, t_2). If the total service rates of streams are proportional to their weights, then their normalized service rates should be equal as the time interval $t_2 - t_1$ goes to infinity. Suppose stream f is backlogged at a set of bricks, denoted as set S, its normalized service rate at S is denoted as $R_{f,S}(t_1, t_2)$, and $R_{f,other}(t_1, t_2)$ denotes its normalized total service rate at all other bricks. $R_f(t_1, t_2) = R_{f,S}(t_1, t_2) + R_{f,other}(t_1, t_2)$. We drop (t_1, t_2) hereafter as we always consider interval (t_1, t_2).

THEOREM 3. *Under* TOTAL-DSFQ, *if during* (t_1, t_2), *streams* $\{f_1, f_2, ...f_n\}$ *are backlogged at a set of bricks* S, *in the order* $R_{f_1,other} \leq R_{f_2,other} \leq ... R_{f_n,other}$, *as* $t_2 - t_1 \rightarrow \infty$, *either* $R_{f_1} = R_{f_2} = ...R_{f_n}$ *or* $\exists k \in \{1, 2, ...n-1\}$, *such that* $R_{f_1} = ...R_{f_k} \leq R_{f_{k+1},other}$ *and* $R_{f_{k+1},S} = ...R_{f_n,S} = 0$.

The intuition of Theorem 3 is as follows. At brick set S, let us first set $R_{f_1,S} = R_{f_2,S} = ... = R_{f_n,S} = 0$ and try to allocate the resources of S. Stream f_1 has the highest priority since its delay is the smallest. Thus the SFQ scheduler will increase $R_{f_1,S}$ until $R_{f_1} = R_{f_2,other}$. Now both f_1 and f_2 have the same total service rate and the same highest priority. Brick set S will then increase $R_{f_1,S}$ and $R_{f_2,S}$ equally until $R_{f_1} = R_{f_2} = R_{f_3,other}$. In the end, either all the streams have the same total service rate, or it is impossible to balance all streams due to the limited service capacity of all bricks in S. In the latter case, the first k streams have equal total service rates, while the remaining streams are blocked for service at S. Intuitively, this is the best we can do with a work-conserving scheduler to equalize normalized service rates.

In Section 3.4 we propose a modification to TOTAL-DSFQ that ensures no stream is blocked at any brick.

3.3.2 Single-client Multi-coordinator

So far we have assumed that a stream requests service through one coordinator only. In many high-end systems, however, it is preferable for high-load clients to distribute their requests among multiple coordinators in order to balance the load on the coordinators. In this section, we discuss the single-client multi-coordinator setting and the corresponding fairness analysis for TOTAL-DSFQ. In summary, we find that TOTAL-DSFQ does engender total proportional sharing in this setting, except in some unusual cases.

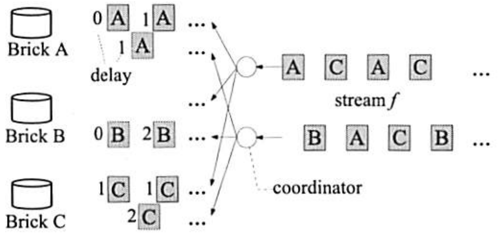


Figure 7: **Effect of multiple coordinators under TOTAL-DSFQ.** Delay value of an individual request is different from Fig. 5, but the total amount of delay remains the same.

We motivate the analysis with an example. First, let us assume that a stream accesses two coordinators in round-robin order and examine the effect on the delay function (7) through the example stream in Fig. 5. The result is displayed in Fig. 7. Odd-numbered requests are processed by the first coordinator and even-numbered requests are processed by the second coordinator. With one coordinator, the three requests to brick A have delay values 0, 2 and 0. With two round-robin coordinators, the delay values of the two requests dispatched by the first coordinator are now 0 and 1; the delay value of the request dispatched by the second coordinator is 1. Thus, although individual request may have delay value different from the case of single coordinator, the total amount of delay remains the same. This is because every virtual request (to other bricks) is counted exactly once.

We formalize this result in Theorem 4 below, which says, essentially, that streams backlogged at a brick receive total proportional service so long as each stream uses a consistent set of coordinators (i.e., the same set of coordinators for each brick it accesses).

Formally, assume stream f sends requests through n coordinators $C_1, C_2, ..., C_n$, and coordinator C_i receives a substream of f denoted as f_i. With respect to brick A, each substream f_i has its $batchcost^{max}_{f_i,A}$. Let us first assume that $batchcost^{max}_{f_i,A}$ is finite for all substreams, i.e., requests to A are distributed among all coordinators.

THEOREM 4. *Assume stream f accesses* n *coordinators such that each one receives substreams* $f_1, ..., f_n$, *respectively, and stream g accesses* m *coordinators with substreams* $g_1, ..., g_m$, *respectively. During any interval* $[t_1, t_2]$ *in which* f *and* g *are both continuously backlogged at brick A, inequality (8) still holds, where*

$$batchcost^{max}_{f,A} = max\{\ batchcost^{max}_{f_1,A}, ... \\ batchcost^{max}_{f_n,A}\} \quad (9)$$

$$batchcost^{max}_{g,A} = max\{\ batchcost^{max}_{g_1,A}, ... \\ batchcost^{max}_{g_m,A}\} \quad (10)$$

An anomalous case arises if a stream partitions the bricks into disjoint subsets and accesses each partition through separate coordinators. In this case, the requests served in one partition will never be counted in the delay of any request to the other partition, and the total service may no longer be proportional to the weight. For example, requests to B in Fig. 7 have smaller delay values than the ones in Fig. 5. This case is unlikely to occur with most load balancing schemes such as round-robin or uniformly random selection of coordinators. Note that the algorithm will still guarantee total proportional sharing if *different* streams use separate coordinators.

More interestingly, selecting randomly among multiple coordinators may smooth out the stream, and result in more uniform delay values. For example, if $batchcost(p^j_{f,A})$ in the original stream is a sequence of i.i.d. (independent, identically distributed) random variables with large variance such that $batchcost^{max}_{f,A}$ might be large, it is not difficult to show that with independently random mapping of each request to a coordinator, $batchcost(p^j_{f_i,A})$ is also a sequence of i.i.d. random variables with the same mean, but the variance decreases as number of coordinators increases. This means that under random selection of coordinators, while the average delay is still the same (thus service rate is the same), the variance in the delay value is reduced and therefore the unfairness bound is tighter. We test this observation through an empirical study later.

3.4 Hybrid Proportional Sharing

Under TOTAL-DSFQ, Theorem 3 tells us that a stream may be blocked at a brick if it gets too much service at other bricks. This is not desirable in many cases. We would like to guarantee a minimum service rate for each stream on every brick so the client program can always make progress. Under the DSFQ framework, i.e., formulae (5-6), this means that the delay must be bounded, using a different delay function than the one used in TOTAL-DSFQ. We next develop a delay function that guarantees a minimum service share to backlogged streams on each brick.

Let us assume that the weights assigned to streams are normalized, i.e. $0 \leq \phi_f \leq 1$ and $\sum_f \phi_f = 1$. Suppose that, in addition to the weight ϕ_f, each stream f is assigned a brick-minimum weight ϕ^{min}_f, corresponding to the minimum service share per brick for the stream.[2] We can then show that the following delay function will guarantee the required minimum service share on each brick for each stream.

$$delay(p^i_{f,A}) = \frac{\phi_f/\phi^{min}_f - 1}{1-\phi_f} * cost(p^i_{f,A}) \quad (11)$$

We can see, for example, that setting $\phi^{min}_f = \phi_f$ yields a delay of zero, and the algorithm then reduces to single brick proportional sharing that guarantees minimum share ϕ_f for stream f, as expected.

By combining delay function (11) with the delay function (7) for TOTAL-DSFQ, we can achieve an algorithm that approaches total proportional sharing while guaranteeing a minimum service level for each stream per brick, as follows.

$$delay(p^i_{f,A}) = \min \left\{ batchcost(p^i_{f,A}) - cost(p^i_{f,A}), \frac{\phi_f/\phi^{min}_f - 1}{1-\phi_f} * cost(p^i_{f,A}) \right\} (12)$$

The DSFQ algorithm using the delay function (12) defines a new algorithm called HYBRID-DSFQ. Since the delay under HYBRID-DSFQ is no greater than the delay in (11), the service rate at every brick is no less than the rate under (11), thus the minimum per brick service share ϕ^{min}_f is still guaranteed. On the other hand, if the amount of service a stream f receives on other bricks between requests to brick A is lower than $(\phi_f/\phi^{min}_f - 1)/(1-\phi_f) * cost(p^i_{f,A})$, the delay function behaves similarly to equation (7), and hence the sharing properties in this case should be similar to TOTAL-DSFQ, i.e., total proportional sharing.

Empirical evidence (in Section 4.3) indicates that HYBRID-DSFQ works as expected for various workloads. However, there are pathological workloads that can violate the total service proportional sharing property. For example, if a stream using two bricks knows its data layout, it can alternate bursts to one brick and then the other. Under TOTAL-DSFQ, the first request in each burst would have received a large delay, corresponding to the service the stream had received on the other brick during the preceding burst, but in HYBRID-DSFQ, the delay is truncated by the minimum share term in the delay function. As a result, the stream receives more service than its weight entitles it to. We believe that this can be resolved by including more history in the minimum share term, but the design and evaluation of such a delay function is reserved to future work.

4 Experimental Evaluation

We evaluate our distributed proportional sharing algorithm in a prototype FAB system [20], which consists of six bricks. Each brick is an identically configured HP ProLiant DL380 server with 2x 2.8GHz Xeon CPU, 1.5GB RAM, 2x Gigabit NIC, and an integrated Smart Array 6i storage controller with four 18G Ultral320, 15K rpm SCSI disks configured as RAID 0. All bricks are running SUSE 9.2 Linux, kernel 2.6.8-24.10-smp. Each brick runs a coordinator. To simplify performance com-

[2] Setting the brick-minimum weights requires knowledge of the client data layouts. We do not discuss this further in the paper.

parisons, FAB caching was turned off, except at the disk level.[3]

The workload generator consists of a number of clients (streams), each running several Postmark [15] instances. We chose Postmark as the workload generator because its independent, randomly-distributed requests give us the flexibility to configure different workloads for demonstration and for testing extreme cases. Each Postmark thread has exactly one outstanding request in the system at any time, accessing its isolated 256MB logical volume. Unless otherwise specified, each volume resides on a single brick and each thread generates random read/write requests with file sizes from 1KB to 16KB. The number of transactions per thread is sufficiently large so the thread is always active when it is on. Other parameters of Postmark are set to the default values.

With our work-conserving schedulers, until a stream is backlogged, its IO throughput increases as the number of threads increases. When it is backlogged, on the other hand, the actual service amount depends on the scheduling algorithm. To simplify calculation, we target throughput (MB/sec) proportional sharing, and define the cost of a request to be its size. Other cost functions such as IO/sec or estimated disk service time could be used as well.

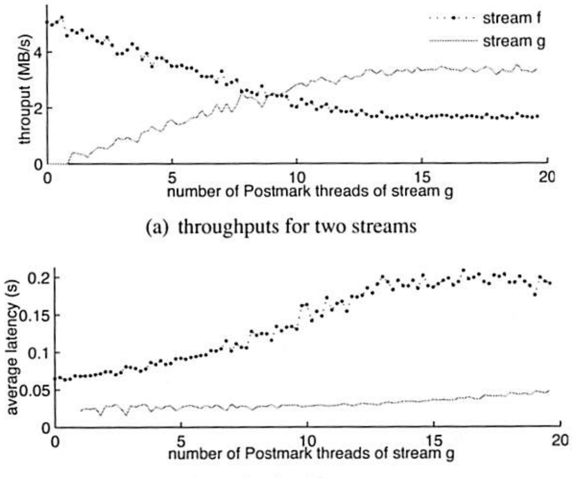
4.1 Single Brick Proportional Sharing

We first demonstrate the effect of TOTAL-DSFQ on two streams reading from one brick. Stream f consistently has 30 Postmark threads, while the number of Postmark threads for stream g is increased from 0 to 20. The ratio of weights between f and g is at 1:2. As the data is not distributed, the delay value is always zero and this is essentially the same as SFQ(D) [12].

Figure 8 shows the performance isolation between the two clients. The throughput of stream g is increasing and its latency is fixed until g acquires its proportional share at around 13 threads. After that, additional threads do not give any more bandwidth but increase the latency. On the other hand, the throughput and latency of stream f are both affected by g. Once g gets its share, it has no further impact on f.

4.2 Total Service Proportional Sharing

Figure 9 demonstrates the effectiveness of TOTAL-DSFQ for two clients. The workload streams have access patterns shown in Fig. 3. We arranged the data layout so that each Postmark thread accesses only one brick. Stream f and stream g both have 30 threads on brick A throughout the experiment, meanwhile, an increasing number of threads from g is processed at brick B. Postmark allows us to specify the maximum size of the random files generated, and we tested the algorithm with workloads using two different maximum random file sizes, 16KB and 1MB.



(a) throughputs for two streams

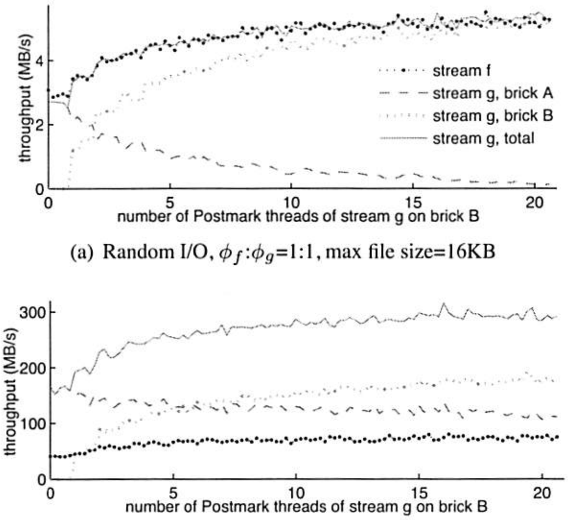
(b) latencies for two streams

Figure 8: **Proportional sharing on one brick.** ϕ_f:ϕ_g=1:2; legend in (a) also applies to (b).

Figure 9(a) shows that as the number of Postmark threads from stream g directed to brick B increases, its throughput from brick B increases, and the share it receives at brick A decreases to compensate. The total throughputs received by streams f and g stay roughly equal throughout. As the stream g becomes more unbalanced between bricks A and B, however, the throughput difference between streams f and g varies more. This can be related to the fairness bound in Theorem 2: as the imbalance increases, so does $batchcost_{g,A}^{max}$, and the bound becomes a little looser. Figure 9(b) uses the same data layout but with a different file size and weight ratio. As g gets more service on B, its throughput rises on B from 0 to 175 MB/s. As a result, the algorithm increases f's share on the shared brick A, and its throughput rises from 40 MB/s to 75 MB/s, while g's throughput on A drops from 160 MB/s to 125 MB/s. In combination the throughput of both streams increases, whole maintaining a 1:4 ratio.

The experiment in Figure 10 has a data layout with dependencies among requests. Each thread in g accesses all three bricks, while stream f accesses one brick only. The resource allocation is balanced when stream g has three or more threads on the shared brick. As g has a RAID-0 data layout, the service rates on the other two bricks are limited by the rate on the shared brick. This experiment shows that TOTAL-DSFQ correctly controls the

[3] Our algorithms focus on the management of storage bandwidth; a full exploration of the management of multiple resources (including cache and network bandwidth) to control end-to-end performance is beyond the scope of this paper.



(a) Random I/O, ϕ_f:ϕ_g=1:1, max file size=16KB

(b) Sequential I/O, ϕ_f:ϕ_g=1:4, max file size=1MB

Figure 9: **Total service proportional sharing.** f's data is on brick A only; g has data on both bricks A and B. As g gets more service on the bricks it does not share with f, the algorithm increases f's share on the brick they do share; thus the total throughputs of both streams increase.

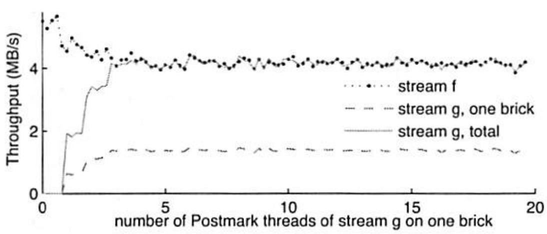
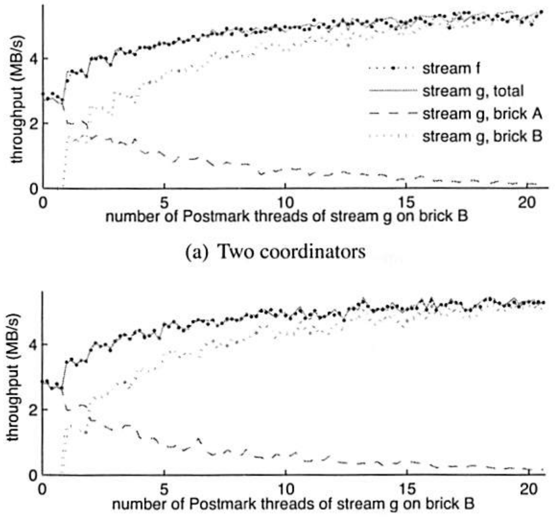


Figure 10: **Total service proportional sharing with striped data.** ϕ_f:ϕ_g=1:1. g has RAID-0 logical volume striping on three bricks; f's data is on one brick only.

total share in the case where requests on different bricks are dependent. We note that in this example, the scheduler could allocate more bandwidth on the shared brick to stream g in order to improve the total system throughput instead of maintaining proportional service; however, this is not our goal.

Figure 11 is the result with multiple coordinators. The data layouts and workloads are the same as in the experiment shown in Figures 3 and 9: two bricks, stream f accesses only one, and stream g accesses both. The only difference is that stream g accesses both bricks A and B through two or four coordinators in round-robin order.

Using multiple coordinators still guarantees proportional sharing of the total throughput. Furthermore, a comparison of Fig. 9, 11(a), and 11(b) indicates that as the number of coordinators increases, the match between the total throughputs received by f and g is closer, i.e., the unfairness bound is tighter. This confirms the observation in Section 3.3.2 that multiple coordinators may smooth out a stream and reduce the unfairness.



(a) Two coordinators

(b) Four coordinators

Figure 11: **Total service proportional sharing with multi-coordinator,** ϕ_f:ϕ_g=1:1

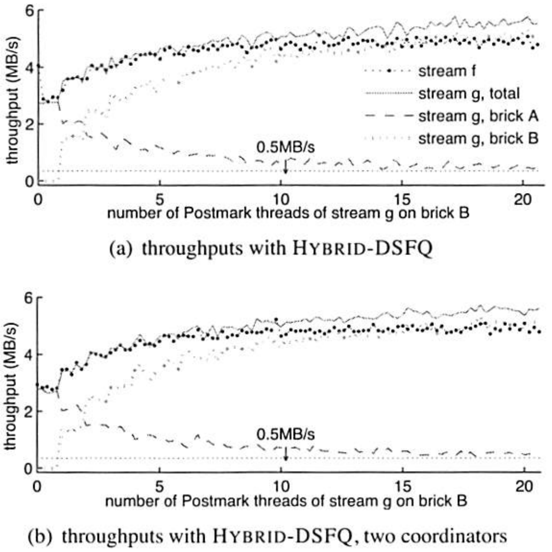
4.3 Hybrid Proportional Sharing

The result of HYBRID-DSFQ is presented in Fig. 12. The workload is the same as in the experiment shown in Figures 3 and 9: stream f accesses brick A only, and stream g accesses both A and B. Streams f and g both have 20 Postmark threads on A, and g has an increasing number of Postmark threads on B. We wish to give stream g a minimum share of 1/12 on brick A when it is backlogged. This corresponds to $\phi_g^{min} = 1/12$; based on Equation 12, the delay function for g is

$$delay(p^i_{g,A}) = \min \; \{ \; batchcost(p^i_{g,A}) - cost(p^i_{g,A}), \\ 10 * cost(p^i_{g,A}) \}$$

Stream f is served on A only and thus the delay is always zero.

With HYBRID-DSFQ, the algorithm reserves a minimum share for each stream, and tries to make the total throughput as close as possible without reallocating the reserved share. For this workload, the service capacity of a brick is approximately 6MB/sec. We can see in Fig. 12(a) that if the throughput of stream g on brick B is less than 4MB, HYBRID-DSFQ can balance the total throughputs of the two streams. As g receives more service on brick B, the maximum delay part in HYBRID-DSFQ takes effect and g gets its minimum share on brick A. The total throughputs are no longer proportional to the assigned weights, but is still reasonably close. Figure 12(b) repeats the experiment with the streams selecting between two coordinators alternately; the workload



(a) throughputs with HYBRID-DSFQ

(b) throughputs with HYBRID-DSFQ, two coordinators

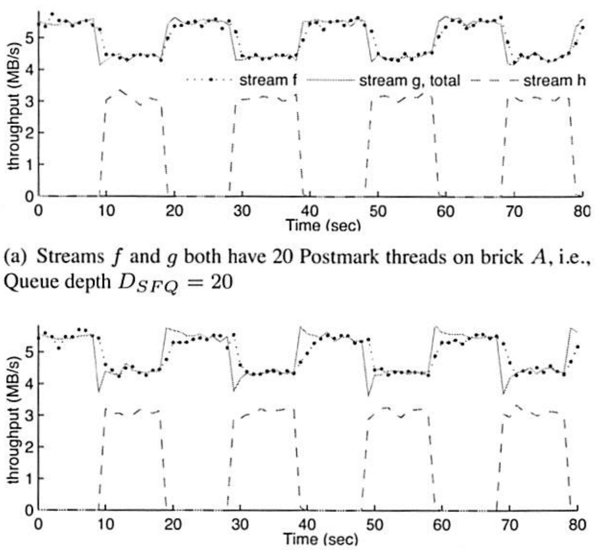
Figure 12: **Two-brick experiment using HYBRID-DSFQ**

and data layout are otherwise identical to the single coordinator experiment. The results indicate that HYBRID-DSFQ works as designed with multiple coordinators too.

4.4 Fluctuating workloads

First we investigate how TOTAL-DSFQ responds to sudden changes in load by using an on/off fluctuating workload. Figure 13 shows the total throughputs of the three streams. Steams f and g are continuously backlogged at brick A and thus the total throughputs are the same. When stream h is on, some bandwidth on brick B is occupied by h (h's service is not proportional to its weight because of insufficient threads it has and thus it is not backlogged on brick B). As a result, g's throughput drops. Then f's throughput follows closely after a second, because part of f's share on A is reallocated to g to compensate its loss on B. Detailed throughputs of g on each brick is not shown on the picture. We also see that as the number of threads (and hence the SFQ depth) increases, the sharp drop in g's throughput is more significant. These experimental observations agree with the unfairness bounds on TOTAL-DSFQ shown in Theorem 2, which increase with the queue depth.

Next we examine the effectiveness of different proportional sharing algorithms through sinusoidal workloads. Both streams f and g access three bricks and overlap on one brick only, brick A. The number of Postmark threads for each stream on each brick is approximately a sinusoidal function with different frequency; see Fig. 14(a). To demonstrate the effectiveness of proportional sharing, we try to saturate brick A by setting the number of threads on it to a sinusoidal function varying from 15 to 35, while thread numbers on other bricks take values from 0 to 10 (not shown in Fig. 14(a)). The result confirms several hypotheses. Figure 14(b) is the result on a standard FAB without any fair scheduling. Not surprisingly, the throughput curves are similar to the thread curves in Fig. 14(a) except when the server is saturated. Figure 14(c) shows that single brick proportional sharing provides proportional service on brick A but not necessarily the total service. At time 250, the service on A is not proportional because g has minimum threads on A and is not backlogged. Figure 14(d) displays the effect of total service proportional sharing. The total service rates match well in general. At times around 65, 100, 150, and 210, the rates deviate because one stream gets too much service on other bricks, and its service on A drops close to zero. Thus TOTAL-DSFQ cannot balance the total service. At time around 230-260, the service rates are not close because stream g is not backlogged, as was the case in Fig. 14(c). Finally, Fig. 14(e) confirms the effect of hybrid proportional sharing. Comparing with Fig. 14(d), HYBRID-DSFQ proportional sharing guarantees minimum share when TOTAL-DSFQ does not, at the cost of slightly greater deviation from total proportional sharing during some periods.



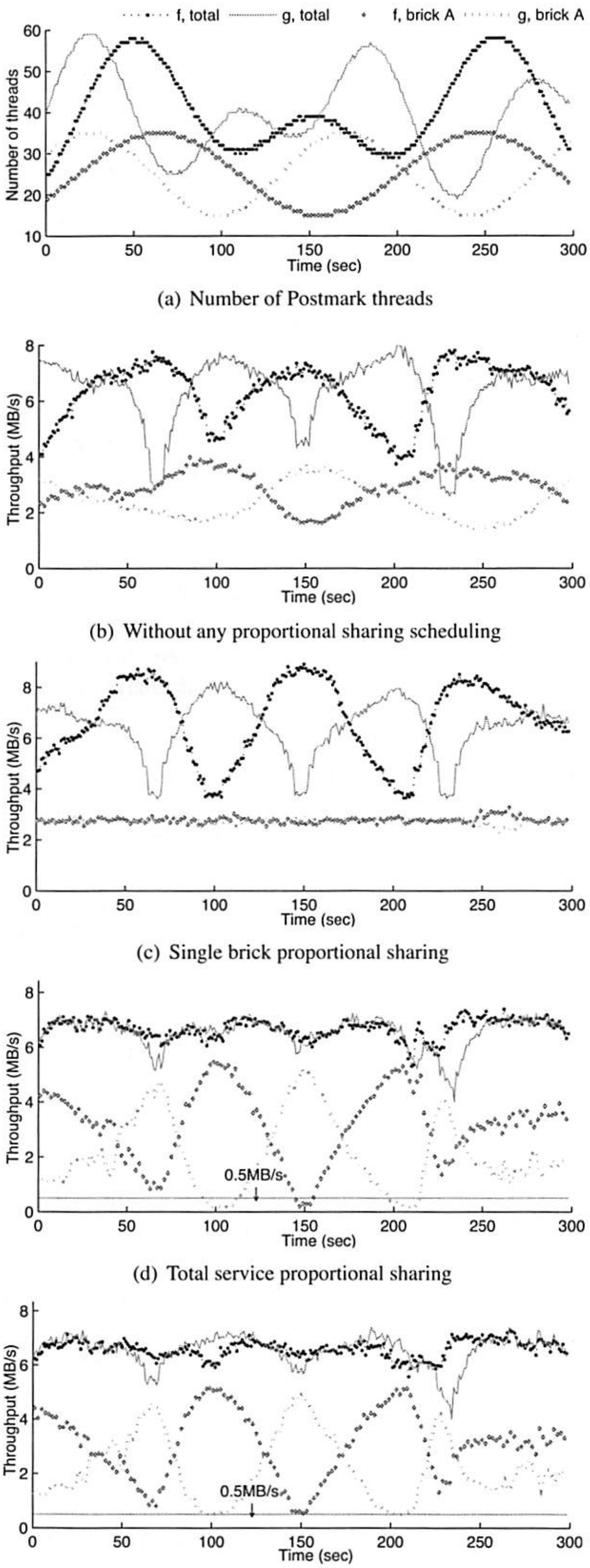
(a) Streams f and g both have 20 Postmark threads on brick A, i.e., Queue depth $D_{SFQ} = 20$

(b) Streams f and g both have 30 threads, $D_{SFQ} = 30$

Figure 13: **Fluctuating workloads.** Streams f and g both have the same number of Postmark threads on brick A, and stream g has 10 additional Postmark threads on brick B. In addition, there is a stream h that has 10 on/off threads on brick B that are repeatedly on together for 10 seconds and then off for 10 seconds. The weights are equal: $\phi_f : \phi_g : \phi_h = 1 : 1 : 1$.

5 Conclusions

In this paper, we presented a proportional-service scheduling framework suitable for use in a distributed storage system. We use it to devise a distributed scheduler that enforces proportional sharing of total service between streams to the degree possible given the workloads. Enforcing proportional total service in a distributed storage system is hard because different clients can access data from multiple storage nodes (bricks) using different, and possibly multiple, access points (coordinators). Thus, there is no single entity that knows the state of all the streams and the service they have received. Our scheduler extends the SFQ(D) [12] algorithm, which was designed as a centralized scheduler. Our scheduler is fully distributed, adds very little communication overhead, has low computational requirements, and is work-conserving. We prove the fairness properties of this scheduler analytically and also show experimental results from an implementation on the FAB distributed storage system that illustrate these properties.



(a) Number of Postmark threads

(b) Without any proportional sharing scheduling

(c) Single brick proportional sharing

(d) Total service proportional sharing

(e) Hybrid proportional sharing

Figure 14: **Sinusoidal workloads,** ϕ_f:ϕ_g=1:1. The legend in (a) applies to all the figures.

We also present examples of unbalanced workloads for which no work-conserving scheduler can provide proportional sharing of the total throughput, and attempting to come close can block some clients on some bricks. We demonstrate a hybrid scheduler that attempts to provide total proportional sharing where possible, while guaranteeing a minimum share per brick for every client. Experimental evidence indicates that it works well.

Our work leaves several issues open. First, we assumed that clients using multiple coordinators load those coordinators equally or randomly; while this is a reasonable assumption in most cases, there may be cases when it does not hold — for example, when some coordinators have an affinity to data on particular bricks. Some degree of communication between coordinators may be required in order to provide total proportional sharing in this case. Second, more work is needed to design and evaluate better hybrid delay functions that can deal robustly with pathological workloads. Finally, our algorithms are designed for enforcing proportional service guarantees, but in many cases, requirements may be based partially on absolute service levels, such as a specified minimum throughput, or maximum response time. We plan to address how this may be combined with proportional sharing in future work.

6 Acknowledgements

Antonio Hondroulis and Hernan Laffitte assisted with experimental setup. Marcos Aguilera, Eric Anderson, Christos Karamanolis, Magnus Karlsson, Kimberly Keeton, Terence Kelly, Mustafa Uysal, Alistair Veitch and John Wilkes provided valuable comments. We also thank the anonymous reviewers for their comments, and Carl Waldspurger for shepherding the paper.

References

[1] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of ser-

vice guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, June 1999.

[2] D. D. Chambliss, G. A. Alvarez, P. Pandey, and D. Jadav. Performance virtualization for large-scale storage systems. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society, Oct. 2003.

[3] H. M. Chaskar and U. Madhow. Fair scheduling with tunable latency: A round-robin approach. *Proceedings of the IEEE*, 83(10):1374–96, 1995.

[4] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM: Symposium proceedings on Communications architectures & protocols*, Sept. 1989.

[5] Z. Dimitrijević and R. Rangaswami. Quality of service support for real-time storage systems. In *International IPSI-2003 Conference*, Oct. 2003.

[6] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Oct. 2003.

[7] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A decentralized algorithm for erasure-coded virtual disks. In *Int. Conf. on Dependable Systems and Networks*, June 2004.

[8] P. Goyal, M. Vin, and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *ACM Transactions on Networking*, 5(5):690–704, 1997.

[9] A. Gulati and P. Varman. Lexicographic QoS scheduling for parallel I/O. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, July 2005.

[10] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[11] L. Huang, G. Peng, and T. cker Chiueh. Multidimensional storage virtualization. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[12] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[13] V. Kanodia, C. Li, A. Sabharwal, B. Sadeghi, and E. Knightly. Distributed priority scheduling and medium access in ad hoc networks. *Wireless Networks*, 8(5):455–466, Nov. 2002.

[14] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proceedings of the 12th International Workshop on Quality of Service (IWQoS)*. IEEE, June 2004.

[15] J. Katcher. Postmark: A new file system benchmark. Technical Report 3022, Network Appliance, Oct. 1997.

[16] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of ASPLOS*. ACM, Oct. 1996.

[17] C. R. Lumb, A. Merchant, and G. A. Alvarez. Façade: virtual storage devices with performance guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, Mar. 2003.

[18] H. Luo, S. Lu, V. Bharghavan, J. Cheng, and G. Zhong. A packet scheduling approach to QoS support in multihop wireless networks. *Mob. Netw. Appl.*, 9(3):193–206, 2004.

[19] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[20] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. Fab: Building distributed enterprise disk arrays from commodity components. In *Proceedings of ASPLOS*. ACM, Oct. 2004.

[21] D. C. Stephens and H. Zhang. Implementing distributed packet fair queueing in a scalable switch architecture. In *Proceedings of INFOCOM*, Mar. 1998.

[22] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proc. of the 9th Intl. Symp. on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS)*. IEEE, Aug. 2001.

[23] N. H. Vaidya, P. Bahl, and S. Gupta. Distributed fair scheduling in a wireless lan. In *MobiCom: Proceedings of the 6th annual international conference on Mobile computing and networking*, Aug. 2000.

[24] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[25] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with cart models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[26] Y. Wang and A. Merchant. Proportional service allocation in distributed storage systems. Technical Report HPL-2006-184, HP Laboratories, Dec. 2006.

[27] W. Wilcke et al. IBM intelligent bricks project—petabytes and beyond. *IBM Journal of Research and Development*, 50(2/3):181–197, Mar. 2006.

[28] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–96, 1995.

[29] J. Zhang, A. Sivasubramaniam, A. Riska, Q. Wang, and E. Riedel. An interposed 2-level I/O scheduling framework for performance virtualization. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2005.

[30] L. Zhang. Virtualclock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems*, 9(2):101–124, 1991.

Argon: performance insulation for shared storage servers

Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, Gregory R. Ganger
Carnegie Mellon University

Abstract

Services that share a storage system should realize the same efficiency, within their share of time, as when they have the system to themselves. The Argon storage server explicitly manages its resources to bound the inefficiency arising from inter-service disk and cache interference in traditional systems. The goal is to provide each service with at least a configured fraction (e.g., 0.9) of the throughput it achieves when it has the storage server to itself, within its share of the server—a service allocated $1/n$th of a server should get nearly $1/n$th (or more) of the throughput it would get alone. Argon uses automatically-configured prefetch/write-back sizes to insulate streaming efficiency from disk seeks introduced by competing workloads. It uses explicit disk time quanta to do the same for non-streaming workloads with internal locality. It partitions the cache among services, based on their observed access patterns, to insulate the hit rate each achieves from the access patterns of others. Experiments show that, combined, these mechanisms and Argon's automatic configuration of each achieve the insulation goal.

1 Introduction

Aggregating services onto shared infrastructures, rather than using separate physical resources for each, is a long-standing approach to reducing hardware and administration costs. It reduces the number of distinct systems that must be managed and allows excess resources to be shared among bursty services. Combined with virtualization, such aggregation strengthens notions such as service outsourcing and utility computing.

When multiple services use the same server, each obviously gets only a fraction of the server's resources and, if continuously busy, achieves a fraction of its peak throughput. But, each service should be able to use its fraction of resources with the same efficiency as when run alone; that is, there should be minimal interference. For resources like the CPU and network, time-sharing creates only minor interference. For the two primary storage system resources — disk head time and cache space — this is not the case.

Disks involve mechanical motion in servicing requests, and moving a disk head from one region to another is slow. The worst-case scenario is when two sequential access patterns become tightly interleaved causing the disk head to bounce between two regions of the disk; performance goes from streaming disk bandwidth to that of a random-access workload. Likewise, cache misses are two orders of magnitude less efficient than cache hits. Without proper cache partitioning, it is easy for one data-intensive service to dominate the cache with a large footprint, significantly reducing the hit rates of other services. Two consequences of disk and cache interference are significant performance degradation and lack of performance predictability. As a result, interference concerns compel many administrators to statically partition storage infrastructures among services.

This paper describes mechanisms that together mitigate these interference issues, insulating[1] services that share a storage system from one another's presence. The goal is to maintain each service's efficiency within a configurable fraction (e.g., 0.9) of the efficiency it achieves when it has the storage server to itself, regardless of what other services share the server. We call this fraction the *R-value*, drawing on the analogy of the thermal resistance measure in building insulation. With an R-value of 1.0, sharing affects the portion of server time dedicated to a service, but not the service's efficiency within that portion. Additionally, insulation increases the predictability of service performance in the face of sharing.

The Argon storage server combines three mechanisms plus automated configuration to achieve the above goal. First, detecting sequential streams and using sufficiently large prefetching/write-back ranges amortizes positioning costs to achieve the configured R-value of streaming bandwidth. Second, explicit cache partitioning prevents any one service from squeezing out others. To maximize the value of available cache space, the space allocated to each service is set to the minimum amount required to achieve the configured R-value of its standalone efficiency. For example, a service that streams large files and exhibits no reuse hits only requires enough cache space to buffer its prefetched data. On-line cache simulation is used to determine the required cache space. Third, disk time quanta are used to separate the disk I/O of services,

[1] We use the term "insulate," rather than "isolate," because a service's performance will obviously depend on the fraction of resources it receives and, thus, on the presence of other services. But, ideally, its efficiency will not.

eliminating interference that arises from workload mixing. The length of each quantum is determined by Argon to achieve the configured R-value, and average response time is kept low by improving overall server efficiency.

Experiments with both Linux and pre-insulation Argon confirm the significant efficiency losses that can arise from inter-workload interference. With its insulation mechanisms enabled, measurements show that Argon mitigates these losses and consistently provides each service with at least the configured R-value of unshared efficiency. For example, when configured with an R-value of 0.95 and simultaneously serving OLTP (TPC-C) and decision support (TPC-H Query 3) workloads, Argon's insulation more than doubles performance for both workloads. Workload combinations that cannot be sufficiently insulated, such as two workloads that require the entire cache capacity to perform well, can be identified soon after an unsupportable workload is added.

This paper makes four main contributions. First, it clarifies the importance of insulation in systems that desire efficient and predictable performance for services that share a storage server. Second, it identifies and experimentally demonstrates the disk and cache interference issues that arise in traditional shared storage. Third, it describes mechanisms that collectively mitigate them. Although each mechanism is known, their application to performance insulation, their inter-relationships, and automated configuration to insulation targets have not been previously explored. Fourth, it experimentally demonstrates their effectiveness in providing performance insulation for shared storage. Overall, the paper shows that Argon provides an important and effective foundation for predictable shared storage.

2 Motivation and related work

Administration costs push for using shared storage infrastructures to support multiple activities/services rather than having separate infrastructures. This section expands on the benefits of shared storage, describes the interference issues that arise during sharing, and discusses previous work on relevant mechanisms and problems. The next section discusses Argon's mechanisms for insulating against such interference.

2.1 Why shared storage?

Many IT organizations support multiple activities/services, such as financial databases, software development, and email. Although many organizations maintain distinct storage infrastructures for each activity/service, using a single shared infrastructure can be much more cost-effective. Not only does it reduce the number of distinct systems that must be purchased and supported, it simplifies several aspects of administration. For example, a given amount of excess resources can easily be made available for growth or bursts in any one of the services, rather than having to be partitioned statically among separate infrastructures (and then moved as needed by administrators). One service's bursts can use excess resources from others that are not currently operating at peak load, smoothing out burstiness across the shared infrastructure. Similarly, on-line spare components can also be shared rather than partitioned, reducing the speed with which replacements must be deployed to avoid possible outages.

2.2 Interference in shared storage

When services share an infrastructure, they naturally will each receive only a fraction of its resources. For non-storage resources like CPU time and network bandwidth, well-established resource management mechanisms can support time-sharing with minimal inefficiency from interference and context switching [4, 26]. For the two primary storage system resources, however, this is not the case. Traditional free-for-all disk head and cache management policies can result in significant efficiency degradations when these resources are shared by multiple services. That is, interleaving multiple access patterns can result in considerably less efficient request processing for each access pattern. Such loss of efficiency results in poor performance for each workload and for the overall system — with fair sharing, for example, each of two services should each achieve at least half the performance they experience when not sharing, but efficiency losses can result in much lower performance for both. Further, the service efficiency is determined by the activities of all workloads sharing a server, making performance unpredictable (even if proportional shares are ensured) and complicating dataset assignment tasks.

Disk head interference: Disk head efficiency can be defined as the fraction of the average disk request's service time spent transferring data to or from the magnetic media. The best case, sequential streaming, achieves disk head efficiency of approximately 0.9, falling below 1.0 because no data is transferred when switching from one track to the next [28]. Non-streaming access patterns can achieve efficiencies well below 0.1, as seek time and rotational latency dominate data transfer time. For example, a disk with an average seek time of 5 ms that rotates at 10,000 RPMs would provide an efficiency of $\approx$0.015 for random-access 8 KB requests (assuming 400 KB per track). Improved locality (e.g., cutting seek distances in half) might raise this value to $\approx$0.02.

Interleaving the access patterns of multiple services can reduce disk head efficiency dramatically if doing so breaks up sequential streaming. This often happens to a sequential access pattern that shares storage with any other access pattern(s), sequential or otherwise. Almost all sequential patterns arrive one request at a time, leaving the disk scheduler with only other services' requests immediately after completing one from the sequential pattern. The scheduler's choice of another service's access will incur a positioning delay and, more germane to this discussion, so will the next request from the sequential pattern. If this occurs repeatedly, the sequential pattern's disk head efficiency can drop by an order of magnitude or more.

Most systems use prefetching and write-back for sequential patterns. Not only can this serve to hide disk access times from applications, it can be used to convert sequences of small requests into fewer, larger requests. The larger requests amortize positioning delays over more data transfer, increasing disk head efficiency if the sequential pattern is interleaved with other requests. Although this helps, most systems do not prefetch aggressively enough to achieve performance insulation [24, 28] — for example, the 64 KB prefetch size common in many operating systems (e.g., BSD and Linux) raises efficiency from ≈0.015 to ≈0.11 when sequential workloads share a disk, which is still far below the streaming bandwidth efficiency of ≈0.9. More aggressive use of prefetching and write-back aggregation is one tool used by Argon for performance insulation.

Cache interference: For some applications, a crucial determinant of storage performance is the cache. Given the scale of mechanical positioning delays, cache hits are several orders of magnitude faster than misses. Also, a cache hit uses no disk head time, reducing disk head interference.

With traditional cache eviction policies, it is easy for one service's workload to get an unfair share of the cache capacity, preventing others from achieving their appropriate cache hit rates. Regardless of which cache eviction policy is used, there will exist certain workloads that fill the cache, due to their locality (recency- or frequency-based) or their request rate. The result can be a significant reduction in the cache hit rate for the other workloads' reads, and thus much lower efficiency if these workloads depend upon the cache for their performance.

In addition to efficiency consequences for reads, unfairness can arise with write-back caching. A write-back cache decouples write requests from the subsequent disk writes. Since writes go into the cache immediately, it is easy for a service that writes large quantities of data to fill the cache with its dirty blocks. In addition to reducing other services' cache hit ratios, this can increase the visible work required to complete each miss — when the cache is full of dirty blocks, data must be written out to create free buffers before the next read or write can be serviced.

2.3 Related work

Argon adapts, extends, and applies some existing mechanisms to provide performance insulation for shared storage servers. This section discusses previous work on these mechanisms and on similar problems in related domains.

Storage resource management: Most file systems prefetch data for sequentially-accessed files. In addition to hiding some disk access delays from applications, accessing data in larger chunks amortizes seeks over larger data transfers when the sequential access pattern is interleaved with others. A key decision is how much data to prefetch [25]. The popular 64 KB prefetch size was appropriate more than a decade ago [21], but is now insufficient [24, 28]. Similar issues are involved in syncing data from the write-back cache, but without the uncertainty of prefetching. Argon complements traditional prefetch/write-back with automated determination of sizes so as to achieve a tunable fraction (e.g., 0.9) of standalone streaming efficiency.

Schindler et al. [27, 28] show how to obtain and exploit underlying disk characteristics to achieve good performance with certain workload mixes. In particular, that work shows that, by accessing data in track-sized track-aligned extents, one can achieve a large fraction of streaming disk bandwidth even when interleaving a sequential workload with other workloads. Such disk-specific mechanisms are orthogonal and could be added to Argon to reduce prefetch/write-back sizes.

Most database systems explicitly manage their caches in order to maximize their effectiveness in the face of interleaved queries [11, 15, 27]. A query optimizer, for example, can use knowledge of query access patterns to allocate for each query just the number of cache pages that it estimates are needed to achieve the best performance for that query [15]. Cao et al. [7, 8] show how these ideas can also be applied to file systems in their exploration of application-controlled file caching. In other work, the TIP [25] system assumes application-provided hints about future accesses and divides the filesystem cache into three partitions that are used for read prefetching, caching hinted blocks for reuse, and caching unhinted blocks for reuse. Argon uses cache partitioning, but with a focus on performance insulation rather than overall performance and without assuming prior knowledge of access patterns. Instead, Argon automatically discovers the necessary cache partition size for each service based on its access pattern.

Resource provisioning in shared infrastructures: Deploying multiple services in a shared infrastructure is a popular concept, being developed and utilized by many. For example, DDSD[39] and Oceano[3] are systems that dynamically assign resources to services as demand fluctuates, based on SLAs and static administrator-set priorities, respectively. Resource assignment is done at the server granularity: at any time, only one service is assigned to any server. Subsequent resource provisioning research ([10, 13, 34, 35]) allows services to share a server, but relies on orthogonal research for assistance with performance insulation.

Most previous QoS and proportional sharing research has focused on resources other than storage. For example, resource containers [4] and virtual services [26] provide mechanisms for controlling resource usage for CPU and kernel resources. Several have considered disk time as a resource to be managed, with two high-level approaches. One approach is to use admission control to admit requests into the storage system according to fair-sharing [17, 36] or explicit performance goals [9, 18, 20, 38]. These systems use feedback control to manage the request rates of each service. They do not, however, do anything to insulate the workloads from one another. Argon complements such approaches by mitigating inefficiency from interference.

A second approach is time-slicing of disk head time. For example, the Eclipse operating system [6] allocates access to the disk in 1/2-second time intervals. Many real-time file systems [1, 12, 19, 23] use a similar approach. With large time slices, applications will be completely performance-insulated with respect to their disk head efficiency, but very high latency can result. Argon goes beyond this approach by automatically determining the lengths of time slices required and by adding appropriate and automatically configured cache partitioning and prefetch/write-back.

Rather than using time-slicing for disk head sharing, one can use a QoS-aware disk scheduler, such as YFQ [5] or Cello [29]. Such schedulers make low-level disk request scheduling decisions that reduce seek times and also maintain per-service throughput balance. Argon would benefit from such a QoS-aware disk scheduler, in place of strict time-slicing, for workloads whose access patterns would not interfere when combined.

3 Insulating from interference

Argon is designed to reduce interference between workloads, allowing sharing with bounded loss of efficiency. In many cases, fairness or weighted fair sharing between workloads is also desired. To accomplish the complementary goals of insulation and fairness, Argon combines three techniques: aggressive amortization, cache partitioning, and quanta-based scheduling. Argon automatically configures each mechanism to reach the configured fraction of standalone efficiency for each workload. This section describes Argon's goals and mechanisms.

3.1 Goals and metrics

Argon provides both insulation and weighted fair sharing. *Insulation* means that efficiency for each workload is maintained even when other workloads share the server. That is, the I/O throughput a service achieves, within the fraction of server time available to it, should be close to the throughput it achieves when it has the server to itself. Argon allows "how close?" to be specified by a tunable *R-value* parameter, analogous to the R-value of thermal insulation, that determines what fraction of standalone throughput each service should receive. So, if the R-value is set to 0.9, a service that gets 50% of a server's time should achieve at least 0.9 of 50% of the throughput it would achieve if not sharing the server. And, that efficiency should be achieved no matter what other services do within the other 50% of the server's time, providing predictability in addition to performance benefits.

Argon's insulation focus is on efficiency, as defined by throughput. While improving efficiency usually reduces average response times, Argon's use of aggressive amortization and quanta-based scheduling can increase variation and worst-case response times. We believe that this is an appropriate choice (Section 5 quantifies our experiences with response time), but the trade-off between efficiency and response time variation is fundamental and can be manipulated by the R-value choice.

Argon focuses on the two primary storage server resources, disk and cache, in insulating a service's efficiency. It assumes that network bandwidth and CPU time will not be bottleneck resources. Given that assumption, a service's share of server time maps to the share of disk time that it receives. And, within that share of server time, a service's efficiency will be determined by what fraction of its requests are absorbed by the cache and by the disk efficiency of those that are not.

Disk efficiency, as discussed earlier, is the fraction of a request's service time spent actually transferring data to or from the disk media. Note that this is not the same as disk utilization—utilization may always be 100% in a busy system, but efficiency may be high or low depending on how much time is spent positioning the disk head for each transfer. Idle time does not affect efficiency. So, a service's disk efficiency during its share of disk time should be within the R-value of its efficiency when not sharing the disk; for a given set of requests, disk efficiency determines disk throughput.

Cache efficiency can be viewed as the fraction of requests absorbed by the cache. Absorbed requests—read hits and dirty block overwrites—are handled by the cache without requiring any disk time. Unlike disk efficiency, cache efficiency cannot be maintained for every mix of services, because each service's cache efficiency is a non-linear function of how much cache space (a finite resource) it receives. To address this, a service will be in one of two states: trial and supported. When first added, a service receives spare cache space and is observed to see how much it needs to achieve the appropriate absorption rate (as described in Section 3.4). If the amount needed fits in that spare cache space, that amount is allocated to the service and the service becomes supported. If not, Argon reports the inability to support the service with full efficiency, allowing an administrator or tool to migrate the dataset to a different server, if desired, or leave it to receive best effort efficiency. Thus, new services may not be able to be supported, but supported services will not have necessary cache space taken from them, thereby maintaining their specified R-value of efficiency.

Argon's fairness focus is on providing explicit shares of server time. An alternative approach, employed in some other systems, is to focus on per-service performance guarantees. In storage systems, this is difficult when mixing workloads because different mixes provide very different efficiencies, which will confuse the feedback control algorithms used in such systems. Argon provides a predictable foundation on which such systems could build. Atop Argon, a control system could manipulate the share allocated to a service to change its performance, with much less concern about efficiency fluctuations caused by interactions with other workloads sharing the system. Exploring this approach is an area of future work.

3.2 Overview of mechanisms

Figure 1 illustrates Argon's high-level architecture. Argon provides weighted fair sharing by explicitly allocating disk time and by providing appropriately-sized cache partitions to each workload. Each workload's cache efficiency is insulated by sizing its cache partition to provide the specified R-value of the absorption rate it would get from using the entire cache. Each workload's disk efficiency is insulated by ensuring that disk time is allotted to clients in large enough quanta so that the majority of time is spent handling client requests, with comparatively minimal time spent at the beginning of a quantum seeking to the workload's first request. To ensure quanta are effectively used for streaming reads without requiring a queue of actual client requests long enough to fill the time, Argon performs aggressive prefetching; to ensure that streaming writes efficiently use the quanta, Argon coalesces them aggressively in write-back cache space.

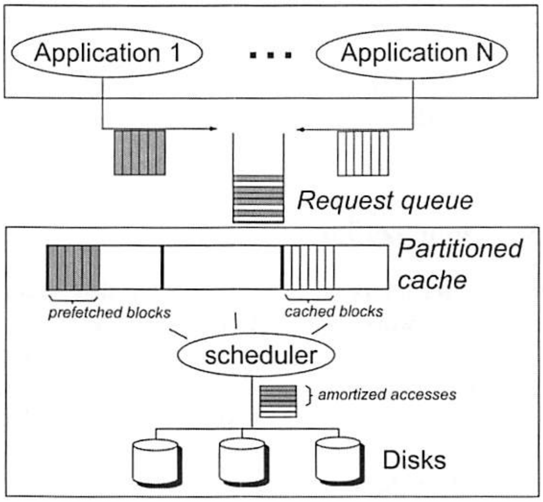


Figure 1: Argon's high-level architecture. Argon makes use of cache partitioning, request amortization, and quanta-based disk time scheduling.

There are four guidelines we follow when combining these mechanisms and applying them to the goals. First, no single mechanism is sufficient to solve all of the obstacles to fairness and efficiency; each mechanism only solves part of the problem. For instance, prefetching improves the performance of streaming workloads, but does not address unfairness at the cache level. Second, some of the mechanisms work best when they can assume properties that are guaranteed by other mechanisms. As an example, both read and write requests require cache space. If there are not enough clean buffers, dirty buffers must first be flushed before a request can proceed. If the dirty buffers belong to a different workload, then some of the first workload's time quantum must be spent performing writes on behalf of the second. Cache partitioning simplifies this situation by ensuring that latent flushes are on behalf of the same workload that triggers them, which makes scheduling easier. Third, a combination of mechanisms is required to prevent unfairness from being introduced. For example, performing large disk accesses for streaming workloads must not starve non-streaming workloads, requiring a scheduler to balance the time spent on each type of workload. Fourth, each mechanism automatically adapts to ensure sufficient insulation based on observed device and workload characteristics and to avoid misconfiguration. For example, the ratio between disk transfer rates and positioning times has changed over time, and full streaming efficiency requires multi-MB prefetches on modern disks instead of the 64–256 KB prefetches of OSes such as Linux and FreeBSD.

3.3 Amortization

Amortization refers to performing large disk accesses for streaming workloads. Because of the relatively high cost of seek times and rotational latencies, amortization is necessary in order to approach the disk's streaming efficiency when sharing the disk with other workloads. However, as is commonly the case, there is a trade-off between efficiency and responsiveness. Performing very large accesses for streaming workloads will achieve the disk's streaming bandwidth, but at the cost of larger variance in response time. Because the disk is being used more efficiently, the average response time actually improves, as we show in Section 5. But, because blocking will occur as large prefetch or coalesced requests are processed, the maximum response time and the variance in response times significantly increase. Thus, the prefetch size should only be as large as necessary to achieve the specified R-value.

In contrast to current file systems' tendency to use 64 KB to 256 KB disk accesses, Argon performs sequential accesses MBs at a time. As discussed in Section 4.2, care is taken to employ a sequential read detector that does not incorrectly predict large sequential access. The exact access size is automatically chosen based on disk characteristics and the configured R-value, using a simple disk model. The average service time for a disk access not in the vicinity of the current head location can be modeled as:

$$S = T_{seek} + T_{rot}/2 + T_{transfer}$$

where S stands for service time, T_{seek} is the average seek time, T_{rot} is the time for one disk rotation, and $T_{transfer}$ is the media transfer time for the data. T_{seek} is the time required to seek to the track holding the starting byte of the data stream. On average, once the disk head arrives at the appropriate track, a request will wait $T_{rot}/2$ before the first byte falls under the head.[2] In contrast to the previous two overhead terms, $T_{transfer}$ represents useful data transfer and depends on the transfer size. In order to achieve disk efficiency of, for example, 0.9, $T_{transfer}$ must be 9 times larger than $T_{seek} + T_{rot}/2$. As shown in Table 1, modern SCSI disks have an average seek time of ≈5 ms, a rotation period of ≈6 ms, and a track size of ≈400 KB. Thus, for the Cheetah 10K.7 SCSI disk to achieve a disk efficiency of 0.9 in a sequential access, $T_{transfer}$ must be $9 * (5\,ms + 6\,ms/2)$ =72 ms. Ignoring head switch time, $\approx 72\,ms/T_{rot} = 12$ tracks must be read, which is 4.8 MB. Each number is higher on a typical SATA drive.

As disks' data densities increase at a much faster rate than improvements in seek times and rotational speeds, aggressive read prefetching and write coalescing grow increasingly important. In particular, the access size of sequential requests required for insulation increases over time. Argon automatically determines the appropriate size for each disk to ensure that it is matched to a server's current devices.

Multiple-MB sequential accesses have two implications. Most significantly, the scheduling quantum for sequential workloads must be sufficiently long to permit large sequential accesses. Consequently, although the average request response time will decrease due to overall increased efficiency, the maximum and variance of the response time usually increases. In addition, the storage server must dedicate multiple-MB chunks of the cache space for use as speed-matching buffers. Both of these limitations are inescapeable if a system is to provide near-streaming disk bandwidth in the presence of multiple workloads, due to mechanical disk characteristics.

So far, we have not distinguished between reads and writes (which are amortized through read prefetching and write coalescing, respectively). From a disk-efficiency standpoint, it does not matter whether one is performing a large read or write request. Write coalescing is straightforward when a client sequentially writes a file into a write-back cache. The dirty cache blocks are sent in large groups (MBs) to the disk. Read prefetching is appropriate when a client sequentially reads a file. As long as the client later reads the prefetched data before it is evicted from the cache, aggressive prefetching increases disk efficiency by amortizing the disk positioning costs.

3.4 Cache partitioning

Cache partitioning refers to explicitly dividing up a server's cache among multiple services. Specifically, if Argon's cache is split into n partitions among services $W_1, ..., W_n$, then W_i's data is only stored in the server's i^{th} cache partition, irrespective of the services' request patterns. Instead of allowing high cache occupancy for some services to arise as an artifact of access patterns and the cache eviction algorithm, cache partitioning preserves a specific fraction of cache space for each service.

[2] Only a small minority of disks have the feature known as *Zero-Latency Access*, which allows them to start reading as soon as the appropriate track is reached and some part of the request is underneath the head (regardless of the position of the first byte) and then reorder the bytes later; this would reduce the $T_{rot}/2$ term.

			Head	Average	Average Sectors		Req. Size for 0.9
Disk	Year	RPM	Switch	Seek	Per Track	Capacity	Efficiency
IBM Ultrastar 18LZX (SCSI)	1999	10000	0.8 ms	5.9 ms	382	18 GB	2.2 MB
Seagate Cheetah X15 (SCSI)	2000	15000	0.8 ms	3.9 ms	386	18 GB	2.5 MB
Maxtor Atlas 10K III (SCSI)	2002	10000	0.6 ms	4.5 ms	686	36 GB	3.4 MB
Seagate Cheetah 10K.7 (SCSI)	2006	10000	0.5 ms	4.7 ms	566	146 GB	4.8 MB
Seagate Barracuda (SATA)	2006	7200	1.0 ms	8.2 ms	1863	250 GB	13 MB

Table 1: SCSI/SATA disk characteristics. Positioning times have not dropped significantly over the last 7 years, but disk density and capacity have grown rapidly. This trend calls for more aggressive amortization.

It is often not appropriate to simply split the cache into equal-sized partitions. Workloads that depend on achieving a high cache absorption rate may require more than $1/n^{th}$ of the cache space to achieve the R-value of their standalone efficiency in their time quantum. Conversely, large streaming workloads only require a small amount of cache space to buffer prefetched data or dirty write-back data. Therefore, knowledge of the relationship between a workload's performance and its cache size is necessary in order to correctly assign it sufficient cache space to achieve the R-value of its standalone efficiency.

Argon uses a three-step process to discover the required cache partition size for each workload. First, a workload's request pattern is traced; this lets Argon deduce the relationship between a workload's cache space and its I/O absorption rate (i.e., the fraction of requests that do not go to disk). Second, a system model predicts the workload's throughput as a function of the I/O absorption rate. Third, Argon uses the specified R-value to compute the required I/O absorption rate (the relationship calculated in step 2), which is then used to select the required cache partition size (the relationship calculated in step 1).

In the first phase, Argon traces a workload's requests. A cache simulator uses these traces and the server's cache eviction policy to calculate the I/O absorption rate for different cache partition sizes. Figure 2 depicts some example cache profiles for commonly-used benchmarks. The total server cache size is 1024 MB. On one hand, the TPC-C cache profile shows that achieving a similar I/O absorption rate to the one achieved with the total cache requires most of the cache space to be dedicated to TPC-C. On the other hand, TPC-H Query 3 can achieve a similar I/O absorption rate to its standalone value with only a fraction of the full cache space. If both the TPC-C workload and TPC-H Query 3 use the same storage server, Argon will give most of the cache space to the TPC-C workload, yet both workloads will achieve similar I/O absorption rates to the ones they obtained in standalone operation.

In the second phase, Argon uses an analytic model to predict the workload's throughput for a specific I/O absorption rate. In the discussion below, we will only consider reads to avoid formula clutter (the details for writes are similar). Let S_i be the average service time, in seconds, of a read request from service i. S_i is modelled as $p_i * S_i^{BUF} + (1 - p_i) * S_i^{DISK}$. Read requests hit the cache with probability p_i and their service time is the cache access time, S_i^{BUF}. The other read requests miss in the cache with probability $1 - p_i$ and incur a service time S_i^{DISK} (write requests similarly can be overwritten in cache or eventually go to disk). p_i is estimated as a function of the workload's cache size, as described in the first step. S_i^{DISK} is continuously tracked per workload, as described in Section 4.4. The server throughput equals $1/S_i$, assuming no concurrency.

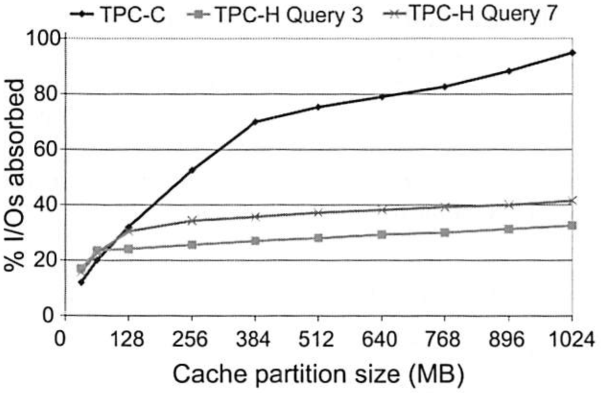


Figure 2: Cache profiles. Different workloads have different working set sizes and access patterns, and hence different cache profiles. The I/O absorption percentage is defined as the fraction of requests that do not go to disk. Such requests include read hits and overwrites of dirty cache blocks. The exact setup for these workloads is described in Section 5.1.

In the final phase, Argon uses the R-value to calculate the required workload throughput when sharing a server as follows:

$$\begin{array}{c}\textit{Throughput required}\\ \textit{in share of time}\end{array} = \textit{(Throughput alone)} \cdot \textit{(R-Value)}$$

A workload must realize nearly its full standalone throughput in its share of time in order to achieve high efficiency. Its actual throughput calculated over the time both it and other workloads are executing, however, may be much less. As an example, suppose a workload receives 10 MB/s of throughput when running alone, and that an R-value of 0.9 is desired. This formula says that the workload must receive at least 9 MB/s in its share of time. (If it is sharing the disk fairly with one other workload, then its overall throughput will be $9 \cdot 50\% =$ 4.5 MB/s.)

Using the second step's analytic model, Argon calculates the minimum I/O absorption rate required for the workload to achieve *Throughput required* during its share of disk time. Then, the minimum cache partition size necessary to achieve the required I/O absorption rate is looked up using the first step's cache profiles. If it is not possible to meet the R-value because of insufficient free cache space, the administrator (or automated management tool) is notified of the best efficiency it could achieve.

3.5 Quanta-based scheduling

Scheduling in Argon refers to controlling when each workload's requests are sent to the disk firmware (as opposed to "disk scheduling," such as SPTF, C-SCAN, or elevator scheduling, which reorders requests for performance rather than for insulation; disk scheduling occurs in the disk's queue and is implemented in its firmware). Scheduling is necessary for three reasons. First, it ensures that a workload receives exclusive disk access, as required for amortization. Second, it ensures that disk time is appropriately divided among workloads. Third, it ensures that the R-value of standalone efficiency for a workload is achieved in its quantum, by ensuring that the quantum is large enough.

There are three phases in a workload's quantum. In the first phase, Argon issues requests that have been queued up waiting for their time slice to begin. If more requests are queued than the scheduler believes will be able to complete in the quantum, only enough to fill the quantum are issued. In the second phase, which only occurs if the queued requests are expected to complete before the quantum is over, the scheduler passes through new requests arriving from the application, if any. The third phase begins once the scheduler has determined that issuing additional requests would cause the workload to exceed its quantum. During this period, the outstanding requests are drained before the next quantum begins.

Inefficiency is introduced by a quanta-based scheduler in two ways. First, if a workload has many outstanding requests, the scheduler may need to throttle the workload and reduce its level of concurrency at the disk in order to ensure it does not exceed its quantum. It is well-known that, for non-streaming workloads, the disk scheduler is most efficient when the disk queue is large. Second, during the third phase, draining a workload's requests also reduces the efficiency of disk head scheduling. In order to automatically select an appropriate quantum size to meet efficiency goals, an analytical lower bound can be established on the efficiency for a given quantum size by modeling these effects for the details (concurrency level and average service time) of the specific workloads in the system. Once a quantum length is established, the number of requests that a particular workload can issue without exceeding its quantum is estimated based on the average service time of its requests, which the scheduler monitors.

Efficiency can also depend upon whether request *mixing* is allowed to happen for non-streaming workloads. Efficiency may be increased by mixing requests from multiple workloads at once, instead of adhering to strict time slices, because this lengthens disk queues. From an insulation standpoint, however, doing so is acceptable only if all clients receive a fair amount of disk time and efficient use of that time. This does not always occur — for instance, some workloads may have many requests "in the pipeline" while others may not. In particular, clients with non-sequential accesses often maintain several outstanding requests at the disk to allow more efficient disk scheduling. Others may not be able to do this; for instance, if the location of the next request depends upon data returned from a preceding request (as when traversing an on-disk data structure), concurrency for that workload is limited. If such workloads are mixed, starvation may occur for the less aggressive workload. Our current design decision has been biased in favor of fairness; we do not allow requests from different workloads to be mixed, instead using strict quanta-based scheduling. This ensures that each client gets exclusive access to the disk during a scheduling quantum, which avoids starvation because active clients' quanta are scheduled in a round-robin manner. In continuing work, we are investigating ways to maintain fairness and insulation while using a mixed-access scheduler.

4 Implementation

We have implemented the Argon storage server to test the efficacy of our performance insulation techniques. Argon is a component in the Ursa Minor cluster-based storage system [2] which exposes an object-based interface [22]. To focus on disk sharing, as opposed to the distributed system aspects of the storage system, we use a single storage server and run benchmarks on the same node, unless otherwise noted.

The techniques of amortization and quanta-based scheduling are implemented on a per-disk basis. Cache partitioning is done on a per-server basis, by default. The design of the system also allows per-disk cache partitioning.

Argon is implemented in C++ and runs on Linux and Mac OS X. For portability and ease-of-development, it is implemented entirely in user-space. Argon stores objects in any underlying POSIX filesystem, with each object stored as a file. Argon performs its own caching; the underlying file system cache is disabled (through `open()`'s `O_DIRECT` option in Linux and `fcntl()`'s `F_NOCACHE` option in Mac OS X). Our servers are battery-backed. This enables Argon to perform write-back caching, by treating all of the memory as NVRAM.

4.1 Distinguishing among workloads

To distinguish among workloads, operations sent to Argon include a client identifier. "Client" refers to a service, not a user or a machine. In our cluster-based storage system, it is envisioned that clients will use sessions when communicating with a storage server; the identifier is an opaque integer provided by the system to the client on a new session. A client identifier can be shared among multiple nodes; a single node can also use multiple identifiers.

4.2 Amortization

To perform read prefetching, Argon must first detect the sequential access pattern to an object. For every object in the cache, Argon tracks a current *run count*: the number of consecutively read blocks. If a client reads a block that is neither the last read block nor one past that block, then the run count is reset to zero. During a read, if the run count is above a certain threshold (4), Argon reads "run count" number of blocks instead of just the requested one. For example, if a client has read 8 blocks sequentially, then the next client read that goes to disk will prompt Argon to read a total of 8 blocks (thus prefetching 7 blocks). Control returns to the client before the entire prefetch has been read; the rest of the blocks are read in the background. The prefetch size grows until the amount of data reaches the threshold necessary to achieve the desired level of disk efficiency; afterwards, even if the run count increases, the prefetch size remains at this threshold.

When Argon is about to flush a dirty block, it checks the cache for any contiguous blocks that are also dirty. In that case, Argon flushes these blocks together to amortize the disk positioning costs. As with prefetching, the write access size is bounded by the size required to achieve the desired level of disk efficiency. Client write operations complete as soon as the block(s) specified by the client are stored in the cache; blocks are flushed to disk in the background (within the corresponding service's quanta).

4.3 Cache partitioning

Recall from Section 3.4 that the cache partitioning algorithm depends on knowledge of the cache profile for a workload. The cache profile provides a relationship between the cache size given to a workload and the expected I/O absorption rate. Argon collects traces of a workload's accesses during the trial phase (when the workload is first added). It then processes those traces using a simulator to predict the absorption rate with hypothetical cache sizes.

The traces collected while a workload is running capture all aspects of its interactions with the cache (cache hits, misses, and prefetches). Such tracing is built in to the storage server, can be triggered on demand (e.g., when workloads change and models need to be updated), and has been shown to incur minimal overheads (5-6%) on foreground workloads in the system [31]. Once sufficient traces for a run are collected, a cache simulator derives the full cache profile for the workload. The simulator does so by replaying the original traces using hypothetical cache sizes and the server's eviction policy. Simulation is used, rather than an analytical model, because cache eviction policies are often complex and system-dependent; we found that they cannot be adequately captured using analytical formulas. We have observed that for cache hits the simulator and real cache manager need similar times to process a request. The simulator is on average three orders of magnitude faster than the real system when handling cache misses (the simulator spends at most 9,500 CPU cycles handling a miss, whereas, on a 3.0 GHz processor, the real system spends the equivalent of about 22,500,000 CPU cycles). The prediction accuracy of the simulator has also been shown to be within 5% [30].

Another implementation issue is dealing with *slack* cache space, the cache space left over after all workloads have taken their minimum share. Currently slack space is distributed evenly among workloads; if a new workload enters the system, the slack space is reclaimed from the other workloads and given to the new workload. This method is very similar to that described by Waldspurger [37] for space reclamation. Other choices are also reasonable, such as assigning the extra space to the workload that would benefit the most, or reserving it for incoming workloads.

4.4 Quanta-based scheduling

Scheduling is necessary to ensure fair, efficient access to the disk. Argon performs simple round-robin time quantum scheduling, with each workload receiving a scheduling quantum. Requests from a particular workload are queued until that workload's time quantum begins. Then, queued requests from that workload are issued, and incoming requests from that workload are passed through to the disk until the workload has submitted what the scheduler has computed to be the maximum number of requests it can issue in the time quantum, or the quantum expires.

The scheduler must estimate how many requests can be performed in the time quantum for a given workload, since average service times of requests may vary between workloads. Initially, the scheduler assigns each request the average rotational plus seek time of the disk. The scheduler then measures the amount of time these requests have taken to derive an average per-request service time for that workload. The automatically-configured scheduling time quantum (chosen based on the desired level of efficiency) is then divided by the calculated average service time to determine the maximum number of requests that will be allowed from that particular workload during its next quantum.

To provide both hysteresis and adaptability in this process, an exponentially weighted moving average is used on the number of requests for the next quantum. As a result of estimation error and changes in the workload over time, the intended time quanta are not always exactly achieved.

Argon does not terminate a quantum until the fixed time length expires. Consequently, workloads with few outstanding requests or with short periods of idle time do not lose the rest of their turn simply because their queue is temporarily empty. Argon does have a policy to deal with situations wherein a time quantum begins but a client has no outstanding requests, however. On one hand, to achieve strict fair sharing, one might reserve the quantum even for an idle workload, because the client might be about to issue a request [14, 16]. On the other hand, to achieve maximum disk utilization, one might skip the client's turn and give the scheduling quantum to the next client which is currently active; if the inactive client later issues a request, it could wait for its next turn or interrupt the current turn. Argon takes a middle approach — a client's scheduling quantum is skipped if the client has been idle for the last k consecutive scheduling quanta. Argon currently leaves k as a manual configuration option (set to 3 by default). It may be possible to automatically select an optimal value for a given workload through trace analysis.

5 Evaluation

This section evaluates the Argon storage server prototype. First, we use micro-benchmarks to show the performance problems arising from storage server interference, and Argon's effectiveness in mitigating them. Micro-benchmarks allow precise control of the workload access patterns and system load. Second, macro-benchmarks illustrate the real-world efficacy of Argon.

5.1 Experimental setup

The machines hosting both the server and the clients have dual Pentium 4 Xeon 3.0 GHz processors with 2 GB of RAM. The disks are Seagate Barracuda SATA disks (see Table 1 for their characteristics). One disk stores the OS, and the other stores the objects (except in one experiment which uses two disks to store objects to focus on the effects of cache sharing). The drives are connected through a 3ware 9550SX controller, which exposes the disks to the OS through a SCSI interface. Both the disks and the controller support command queuing. All computers run the Debian "testing" distribution and use Linux kernel version 2.4.22.

Unless otherwise mentioned, all experiments are run three times, and the average is reported. Except where noted, the standard deviation is less than 5% of the average.

5.2 Micro-benchmarks

This section illustrates micro-benchmark results obtained using both Linux and Argon. These experiments underscore the need for performance insulation and categorize the benefits that can be expected along the three axes of amortization, cache partitioning, and quanta-based scheduling.

Micro-benchmarks are run on a single server, accessing Argon using the object-based interface. In each experiment, objects are stored on the server and are accessed by clients running on the same server (to emphasize the effects of disk sharing, rather than networking effects). Each object is 56 GB in size, a value chosen so that all of the disk traffic will be contained in the highest-performance zone of the disk.[3] The objects are written such that each is fully contiguous on disk. While the system is configured so that no caching of data will occur at the operating system level, the experiments are performed in a way that ensures all of the metadata (e.g., inodes and indirect blocks) needed to access the objects is

[3] Disks have different zones, with only one zone experiencing the best streaming performance. To ensure that the effects of performance insulation are not conflated with such disk-level variations, it is necessary to contain experiments within a single zone of the disk.

cached, to concentrate solely on the issue of data access. In experiments involving non-streaming workloads, unless otherwise noted, the block selection process is configured to choose a uniformly distributed subset of the blocks across the file. The aggregate size of this subset is chosen relative to the cache size to achieve the desired absorption rate.[4]

Amortization: Figure 3(a) shows the performance degradation due to insufficient request amortization in Linux. Two streaming read workloads, each of which receives a throughput of approximately 63 MB/s when running alone, do not utilize the disk efficiently when running together. Instead, each receives a ninth of its unshared performance, and the disk is providing, overall, only one quarter of its streaming throughput. Disk accesses for each of the workloads end up being 64 KB in size, which is not sufficient to amortize the cost of disk head movement when switching between workloads, even though Linux does perform prefetching.

Figure 3(b) shows the effect of amortization in Argon. The version of Argon without performance insulation has similar problems to Linux. However, by performing aggressive amortization (in this case, using a prefetch size of 8 MB, which corresponds, for the disk being used, to an R-value of 0.9), streaming workloads better utilize the disk and achieve higher throughput — both workloads receive nearly half of their performance when running alone, and the disk is providing nearly its full streaming bandwidth.

Figure 4 shows the CDF (cumulative distribution function) of response time for one of the streaming read workloads in each scenario. (The other workload exhibits a virtually identical distribution because the workloads are identical in this experiment.) The three curves depict response times for the cases of the sequential workloads running alone, together without prefetching, and together with prefetching. The value on the y-axis indicates what fraction of requests experienced, at most, the corresponding response time on the x-axis (which is shown in log scale). For instance, when running alone, approximately 80% of the requests had a response time not exceeding 2 ms. Without performance insulation, each sequential workload not only suffered a loss of throughput, but also an increase in average response times; approximately 85% of the requests waited for 25–29 ms. With prefetching enabled, more than 97% of the requests experienced a response time of less than 1 ms, with many much less.[5] Because some requests must wait in a queue for their workload's time slice to begin, however, a small number (≈2.4%) had response times above 95 ms. This increases the variance in response time, while the mean and median response times decrease.

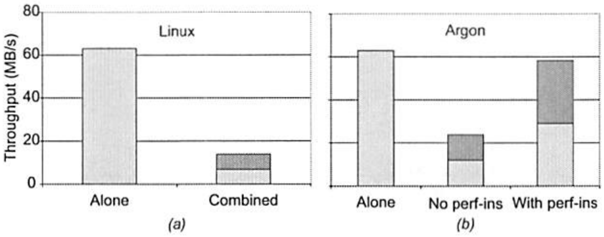


Figure 3: Throughput of two streaming read workloads in Linux and Argon.

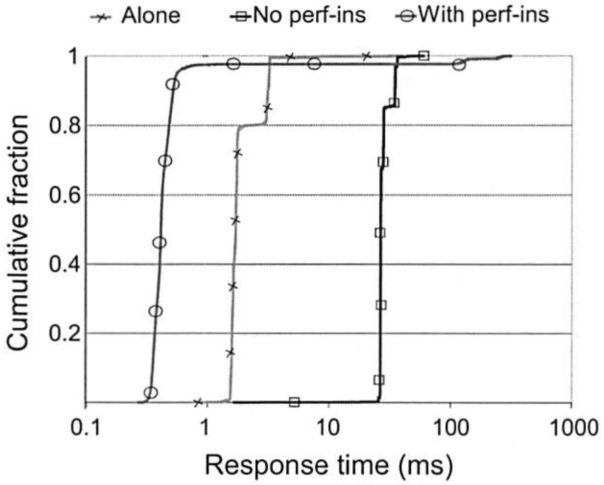


Figure 4: Response time CDFs. When running alone, the average of response times is 2.0 ms and the standard deviation is 1.17 ms. When the two workloads are mixed without performance insulation, the average for each is 28.2 ms and the standard deviation is 3.2 ms. When using performance insulation, the average is 4.5 ms and the standard deviation is 27 ms.

Cache partitioning: Figure 5(a) shows the performance degradation due to cache interference in Linux. A streaming workload (Workload 1), when run together with a non-streaming workload (Workload 2) with a cache absorption rate of 50%, degrades the performance of the non-streaming workload. To focus on only the cache partitioning problem, both workloads share the same cache, but go to separate disks. Because of its much higher throughput, the streaming workload evicts nearly all of the blocks belonging to the non-streaming workload. This causes the performance of the latter to decrease to approximately what it would receive if it had no cache hits at all — its performance drops from 3.9 MB/s to 2.3 MB/s, even though only the cache, and not the disk, is being shared. We believe that the small decrease

[4] One alternative would be to vary the file size to control absorption rates, but this would also affect the disk seek distance, adding another variable to the experiments.

[5] In fact, the response times for many requests improve beyond the standalone case because no prefetching was being performed in the original version of Argon.

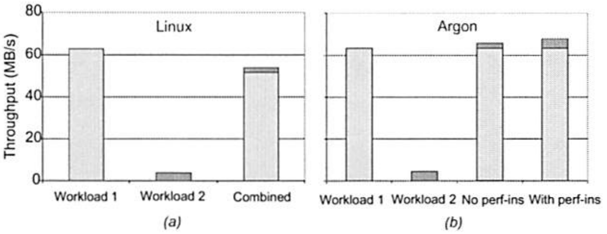


Figure 5: Effects of cache interference in Linux and Argon. The standard deviation is at most 0.55 MB/s for all Linux runs and less than 5% of the average for the Argon runs.

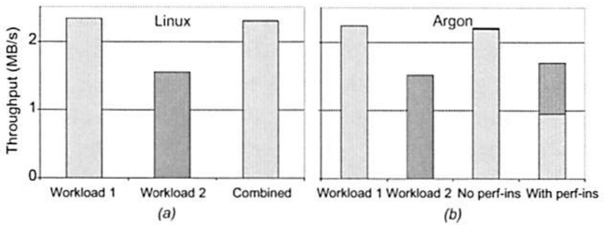


Figure 6: Need for request scheduling in Linux and Argon. The standard deviation is at most 0.01 MB/s for all Linux runs and at most 0.02 MB/s for all Argon runs.

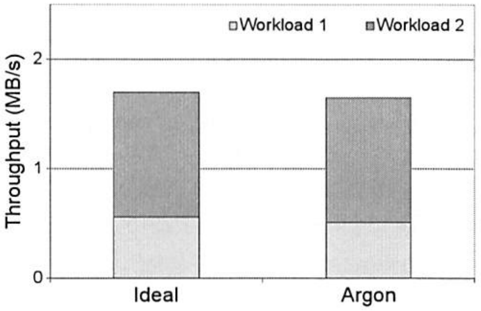


Figure 7: Scheduling support for two random-access workloads. With the same workloads as Figure 6(b), scheduling can be adjusted so that Workload 2 gets 75% of the server time.

in the streaming workload's performance is an artifact of a system bottleneck.

Figure 5(b) shows the effect when the same workloads run on Argon. The bar without performance insulation shows the non-streaming workload combined with the streaming workload. In that case, the performance the non-streaming workload receives equals the performance of a non-streaming workload with a 0% absorption rate. By adding cache partitioning and using the cache simulator to balance cache allocations (setting the desired R-value to 0.9, the simulator decides to give nearly all of the cache to the non-streaming workload), Workload 2 gets nearly all of its standalone performance.

Quanta-based scheduling: Figure 6(a) shows the performance degradation due to unfair scheduling of requests in Linux. Two non-streaming workloads, one with 27 requests outstanding (Workload 1) and one with just 1 request outstanding (Workload 2), are competing for the disk. When run together, the first workload overwhelms the disk queue and starves the requests from the second workload. Hence, the second workload receives practically no service from the disk at all.

Figure 6(b) shows the effect of quanta-based disk time scheduling in Argon. The version of Argon with performance insulation disabled had similar problems to Linux. However, by adding quanta-based scheduling with 140 ms time quanta (which achieves an R-value of 0.9 for the disk and workloads being used), the two non-streaming workloads each get a fair share of the disk. Average response times for Workload 1 increased by $\approx$2.3 times and average response times for Workload 2 decreased by $\approx$37.1 times compared to their uninsulated performance. Both workloads received slightly less than 50% of their unshared throughput, exceeding the R = 0.9 bound.

Proportional scheduling: Figure 7 shows that the sharing of an Argon server need not be fair; the proportion of performance assigned to different workloads can be adjusted to meet higher-level goals. In the experiment, the same workloads as in Figure 6(b) are shown, but the requirement is that the workload with one request outstanding (Workload 2) receive 75% of the server time, and the workload with 27 requests outstanding (Workload 1) receive only 25%; quanta sizes are proportionally sized to achieve this. Amortization and cache partitioning can similarly be adapted to use weighted priorities.

Combining sequential and random workloads: Table 2 shows the combination of the amortization and scheduling mechanisms when a streaming workload shares the storage server with a non-streaming workload. To focus on just the amortization and scheduling effects, the non-sequential workload does not hit in cache at all. Without performance insulation, the workloads receive 2.2 MB/s and 0.55 MB/s respectively. With performance insulation they receive 31.5 MB/s and 0.68 MB/s, well within R = 0.9 of standalone efficiency, as desired.

Figure 8 shows the CDF of response times for both workloads. The sequential workload, shown in Figure 8(a), exhibits the same behavior shown in Figure 4 and discussed earlier. As before, the variance and maximum of response times increase while the mean and median decrease. The random workload is shown in Figure 8(b). Running alone, it had a range of response times, with none exceeding 26 ms. The 90^{th} percentile was at 13.7 ms. Virtually all values were above 3 ms. Running together with the sequential workload, response times in-

Scenario		Throughput
Alone	Workload 1 (S)	63.5 MB/s
	Workload 2 (R)	1.5 MB/s
Combined (no perf-ins.)	Workload 1 (S) Workload 2 (R)	2.2 MB/s 0.55 MB/s
Combined (with perf-ins.)	Workload 1 (S) Workload 2 (R)	31.5 MB/s 0.68 MB/s

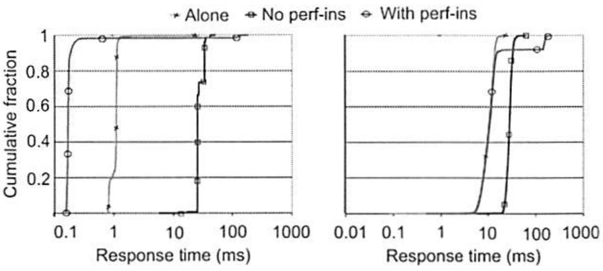
Table 2: Amortization and scheduling effects in Argon. Performance insulation results in much higher efficiency for both workloads. Standard deviation was less than 6% for all runs.

creased; they ranged from 6–60 ms with the 90th percentile at 33 ms. Once aggressive prefetching was enabled for the sequential workload, the bottom 92% of response times for the random workload ranged from 3–24.5 ms. The remainder were above 139 ms, resulting in a lower mean and median, but higher variance.

Scaling number of workloads: Figure 9 shows the combined effect of all three techniques on four workloads sharing a storage server. Workload 1 is the streaming workload used in the previous experiments. Workload 2 is a uniformly random workload with a standalone cache absorption rate of 12.5%. Workload 3 is a microbenchmark that mimics the behavior of TPC-C (with a non-linear cache profile similar to that shown in Figure 2). Workload 4 is a uniformly random workload with zero cache absorption rate. All four workloads get within the desired R-value (0.9) of standalone efficiency when sharing the storage server.

Adjusting sequential access size: Figure 10 shows the effect of prefetch size on throughput. Two streaming workloads, each with an access size of 64 KB, were run with performance insulation. The performance each of them receives is similar, hence we only show the throughput of one of them. In isolation, each of these workloads receives approximately 62 MB/s, hence the ideal scenario would be to have them each receive 31 MB/s when run together. This graph shows that the desired throughput is achieved with a prefetch size of at least 32 MB, and that R = 0.9 can be achieved with 8 MB prefetches. We observed that further increases in prefetch size do not improve, or degrade, performance significantly.

Adjusting scheduling quantum: Figure 11 shows the result of a single-run experiment intended to measure the effect of the scheduling quantum (or the amount of disk time scheduled for one workload before moving on to the other workloads) on throughput. For simplicity, we show quanta measured in number of requests for this figure, rather than in terms of time — since different workloads may have different average service times, the scheduler actually schedules in terms of time, not number of requests. Two non-streaming workloads are running insulated from each other. We only show the throughput of one of them. In isolation, the workload shown receives approximately 2.23 MB/s, hence the ideal scenario would be to have it receive 1.11 MB/s when run together with the other. This graph shows that the desired throughput is achieved with a scheduling quantum of at least 128 requests, and that R = 0.9 can be achieved with one of 32. We observed that further increases in quantum size do not improve, or degrade, performance significantly.



(a) Workload 1 (Sequential)

(b) Workload 2 (Random)

Figure 8: Response time CDFs. The standard deviation of response time for the sequential (a) and random-access workloads (b) when they run alone is 0.316 ms and 2.72 ms respectively. The random-access workload's average response time is 10.3 ms. When the two workloads are mixed without performance insulation, the standard deviation of their response times is 4.16 ms and 4.01 ms respectively. The random-access workload's average response time is 28.2 ms. When using performance insulation the standard deviation is 15.87 ms and 39.3 ms respectively. The random-access workload's average response time is 21.9 ms.

5.3 Macro-benchmarks

To explore Argon's techniques on more complex workloads, we ran TPC-C (an OLTP workload) and TPC-H (a decision support workload) using the same storage server. The combined workload is representative of realistic scenarios when data mining queries are run on a database while transactions are being executed. The goal of the experiment is to measure the benefit each workload gets from performance insulation when sharing the disk.

Each workload is run on a separate machine, and communicates with the Argon storage server through an NFS server that is physically co-located with Argon and uses its object-based access protocol.

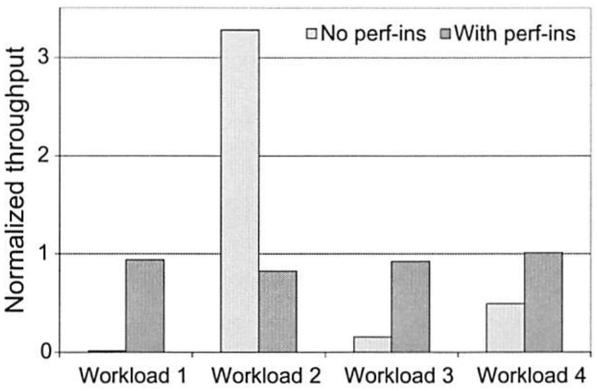


Figure 9: Four workloads sharing a storage server. The normalization is done with respect to the throughput each workload receives when running alone, divided by four.

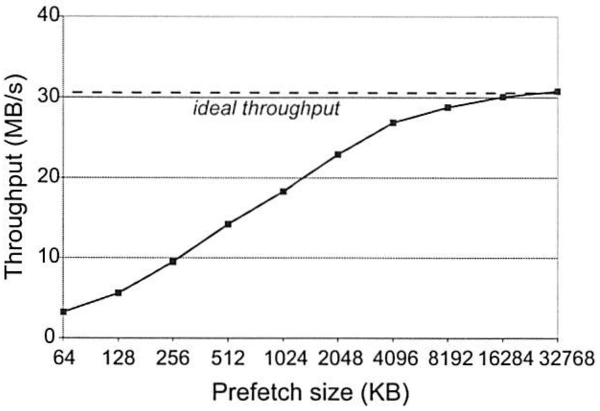


Figure 10: Effect of prefetch size on throughput.

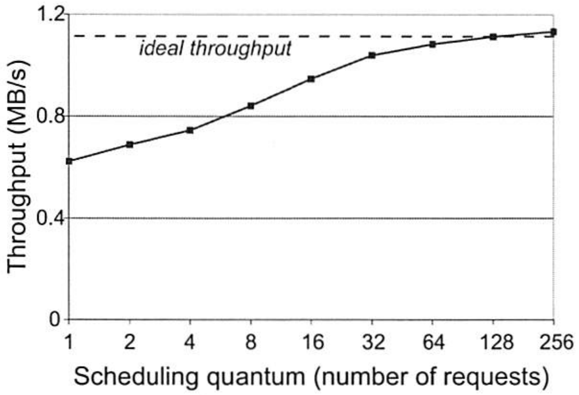


Figure 11: Effect of scheduling quantum on throughput.

TPC-C workload: The TPC-C workload mimics an on-line database performing transaction processing [32]. Transactions invoke 8 KB read-modify-write operations to a small number of records in a 5 GB database. The performance of this workload is reported in transactions per minute (tpm). The cache profile of this workload is shown in Figure 2.

TPC-H workload: TPC-H is a decision-support benchmark [33]. It consists of 22 different queries, and two batch update statements. Each query processes a large portion of the data in streaming fashion in a 1 GB database. The cache profile of two (arbitrarily) chosen queries from this workload are shown in Figure 2.

Figure 12 shows the results: without performance insulation, the throughput of both benchmarks degrades significantly. With an R-value of 0.95, Argon's insulation significantly improves the performance for both workloads. Figure 13 examines the run with TPC-H Query 3 more closely. This figure shows how much each of the three techniques, scheduling (S), amortization (A), and cache partitioning (CP) contribute to maintaining the desired efficiency.

6 Conclusions and future work

Storage performance insulation can be achieved when services share a storage server. Traditional disk and cache management policies do a poor job, allowing interference among services' access patterns to significantly reduce efficiency (e.g., by factor of four or more). Argon combines and automatically configures prefetch/write-back, cache partitioning, and quanta-based disk time scheduling to provide each service with a configurable fraction (the R-value; e.g., 0.9) of the efficiency it would receive without competition. So, with fair sharing, each of n services will achieve no worse than R/n of its standalone throughput. This increases both efficiency and predictability when services share a storage server.

Argon provides a strong foundation on which one could build a shared storage utility with performance guarantees. Argon's insulation allows one to reason about the throughput that a service will achieve, within its share, without concern for what other services do within their share. Achieving performance guarantees also requires an admission control algorithm for allocating shares of server resources, which can build on the Argon foundation. In addition, services that cannot be insulated from one another (e.g., because they need the entire cache) or that have stringent latency requirements must be separated. Argon's configuration algorithms can identify the former and predict latency impacts so that the control system can place such services' datasets on distinct storage servers. Our continuing work is exploring the design of such a control system, as well as approaches for handling workload changes over time.

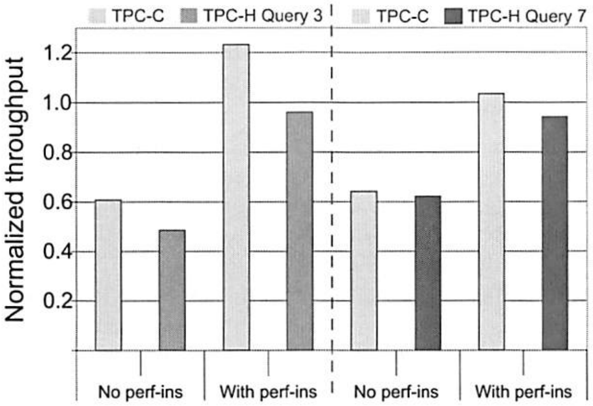


Figure 12: TPC-C and TPC-H running together. TPC-C shown running with TPC-H Query 3 and then with TPC-H Query 7. The normalized throughput with and without performance insulation in shown. The normalization is done with respect to the throughput each workload receives when running alone, divided by two.

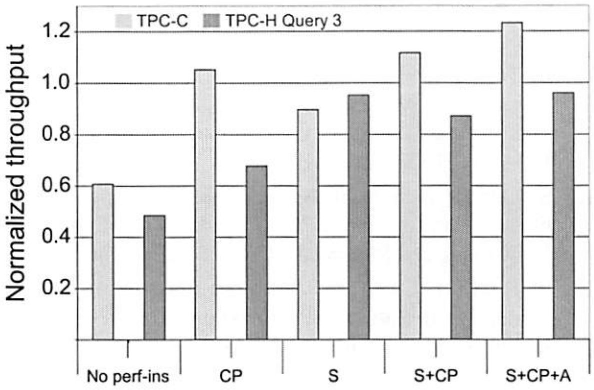


Figure 13: All three mechanisms are needed to achieve performance insulation. The different techniques are examined in combination. "CP" is cache partitioning, "S" is scheduling, "A" is amortization. Argon uses all of them in combination. The normalization is done with respect to the throughput each workload receives when running alone, divided by two.

Acknowledgements

We thank Gregg Economou, Michael Stroucken, Chuck Cranor, and Bill Courtright for assistance in configuring hardware, Craig Soules, John Strunk, Amin Vahdat (our shepherd) and the many anonymous reviewers for their feedback. We thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Oracle, Panasas, Seagate, and Symantec) for their interest, insights, feedback, and support. We also thank Intel, IBM, Network Appliances, Seagate, and Sun for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grants #CNS-0326453 and #CCF-0621499, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by the Army Research Office, under agreement number DAAD19–02–1–0389. Matthew Wachs is supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense.

References

[1] R. Abbott and H. Garcia-Molina. *Scheduling real-time transactions with disk-resident data*. CS-TR-207-89. Department of Computer Science, Princeton University, February 1989.

[2] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. Conference on File and Storage Technologies, pages 59–72. USENIX Association, 2005.

[3] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA Based Management of a Computing Utility. IM – IFIP/IEEE International Symposium on Integrated Network Management, pages 855–868. IFIP/IEEE, 2001.

[4] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. Symposium on Operating Systems Design and Implementation, pages 45–58. ACM, Winter 1998.

[5] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. IEEE International Conference on Multimedia Computing and Systems, pages 400–405. IEEE, 1999.

[6] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. USENIX Annual Technical Conference, pages 235–246. USENIX Association, 1998.

[7] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. Symposium on Operating Systems Design and Implementation, pages 165–177. Usenix Association, 14–17 November 1994.

[8] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. Summer USENIX Technical Conference, pages 171–182, 6–10 June 1994.

[9] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. Symposium on Reliable Distributed Systems, pages 109–118. IEEE, 2003.

[10] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centres. ACM Symposium on Operating Sys-

tem Principles. Published as *Operating Systems Review*, **35**(5):103–116, 2001.

[11] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. International Conference on Very Large Databases, pages 127–141, 21–23 August 1985.

[12] S. J. Daigle and J. K. Strosnider. Disk scheduling for multimedia data streams. SPIE Conference on High-Speed Networking and Multimedia Computing, February 1994.

[13] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. USITS - USENIX Symposium on Internet Technologies and Systems. USENIX Association, 2003.

[14] L. Eggert and J. D. Touch. Idletime scheduling with preemption intervals. ACM Symposium on Operating System Principles, pages 249–262. ACM Press, 2005.

[15] C. Faloutsos, R. Ng, and T. Sellis. Flexible and adaptable buffer management-techniques for database-management systems. *IEEE Transactions on Computers*, **44**(4):546–560, April 1995.

[16] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. ACM Symposium on Operating System Principles. Published as *Operating System Review*, **35**(5):117–130. ACM, 2001.

[17] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 37–48. ACM Press, 2004.

[18] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance Isolation and Differentiation for Storage Systems. International Workshop on Quality of Service, pages 67–74. IEEE, 2004.

[19] P. Lougher and D. Shepherd. The design of a storage server for continuous media. *Computer Journal*, **36**(1):32–42. IEEE, 1993.

[20] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: virtual storage devices with performance guarantees. Conference on File and Storage Technologies, pages 131–144. USENIX Association, 2003.

[21] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. USENIX Annual Technical Conference, pages 33–43. USENIX, 1991.

[22] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based Storage. *Communications Magazine*, **41**(8):84–90. IEEE, August 2003.

[23] A. Molano, K. Juvva, and R. Rajkumar. Real-time filesystems. Guaranteeing timing constraints for disk accesses in RT-Mach. Proceedings Real-Time Systems Symposium, pages 155–165. IEEE Comp. Soc., 1997.

[24] A. E. Papathanasiou and M. L. Scott. Aggressive prefetching: an idea whose time has come. Hot Topics in Operating Systems, 2005.

[25] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **29**(5):79–95, 1995.

[26] J. Reumann, A. Mehra, K. G. Shin, and D. Kandlur. Virtual services: a new abstraction for server consolidation. USENIX Annual Technical Conference, pages 117–130. USENIX Association, 2000.

[27] J. Schindler, A. Ailamaki, and G. R. Ganger. Lachesis: robust database storage management based on device-specific performance characteristics. International Conference on Very Large Databases. Morgan Kaufmann Publishing, Inc., 2003.

[28] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. Conference on File and Storage Technologies, pages 259–274. USENIX Association, 2002.

[29] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. Published as *Performance Evaluation Review*, **26**(1):44–55, 1998.

[30] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. International conference on autonomic computing, 2006.

[31] E. Thereska, B. Salmon, J. Strunk, M. Wachs, Michael-Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 2006.

[32] Transaction Processing Performance Council. TPC Benchmark C, December 2002. `http://www.tpc.org/tpcc/`.

[33] Transaction Processing Performance Council. TPC Benchmark H, December 2002. `http://www.tpc.org/tpch/`.

[34] B. Urgaonkar and P. Shenoy. Sharc: Managing CPU and Network Bandwidth in Shared Clusters. *IEEE Transactions on Parallel and Distributed Systems*, **15**(1):2–17. IEEE, 01–01 January 2004.

[35] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. Symposium on Operating Systems Design and Implementation, pages 239–254. ACM Press, 2002.

[36] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: sharing and isolation in shared memory multiprocessors. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **33**(11):181–192, November 1998.

[37] C. A. Waldspurger. Memory resource management in VMWare ESX server. Symposium on Operating Systems Design and Implementation, 2002.

[38] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy. Zygaria: Storage Performance as a Managed Resource. RTAS – IEEE Real-Time and Embedded Technology and Applications Symposium, pages 125–134, 2006.

[39] H. Zhu, H. Tang, and T. Yang. Demand-driven Service Differentiation in Cluster-based Network Servers. IEEE INFOCOM, pages 679–688. IEEE, 2001.

Strong Accountability for Network Storage

Aydan R. Yumerefendi and Jeffrey S. Chase
Duke University
{aydan,chase}@cs.duke.edu

Abstract

This paper presents the design, implementation, and evaluation of CATS, a network storage service with strong accountability properties. A CATS server annotates read and write responses with evidence of correct execution, and offers audit and challenge interfaces that enable clients to verify that the server is faithful. A faulty server cannot conceal its misbehavior, and evidence of misbehavior is independently verifiable by any participant. CATS clients are also accountable for their actions on the service. A client cannot deny its actions, and the server can prove the impact of those actions on the state views it presented to other clients.

Experiments with a CATS prototype evaluate the cost of accountability under a range of conditions and expose the primary factors influencing the level of assurance and the performance of a strongly accountable storage server. The results show that strong accountability is practical for network storage systems in settings with strong identity and modest degrees of write-sharing. The accountability concepts and techniques used in CATS generalize to a broader class of network services.

1 Introduction

A system is *accountable* if it provides a means to detect and expose misbehavior by its participants. Accountability provides powerful incentives to promote cooperation and discourage malicious and incorrect behavior.

This paper proposes and evaluates system support for *strongly accountable* storage protocols and services. A system is strongly accountable if it provides a means for each participant to determine for itself if others are behaving correctly, without trusting assertions of misbehavior by another participant who may itself be compromised.

To illustrate strong accountability properties and the means to provide them for shared storage, we present CATS, a *certified accountable tamper-evident storage* service. CATS is a rudimentary network storage service: it enables clients to read and write a shared directory of objects maintained by a CATS server. CATS provides clients with the means to verify that the server executes writes correctly, and that read responses are correct given the sequence of valid writes received at the server. Crucially, strong accountability of the server also extends to the clients: a correct server can prove that its state resulted from actions taken by specific clients. Clients cannot deny or repudiate their operations on a strongly accountable server, or the impact of those operations on the shared state. The CATS network storage service is based on a generic state storage toolkit. Our intent is that the CATS toolkit can serve as a substrate for a range of accountable services with richer semantics.

Various notions of accountability and end-to-end trust appear in many previous systems (see Section 6). Our approach is focused on assuring *semantically* faithful behavior, rather than performance accountability [15]. For example, a compromised CATS server can deny service, but it cannot misrepresent the shared state without risk of exposure that is provable to all participants. In this respect, CATS offers a strong form of tamper-evidence [17] at the server. More generally, strong accountability may be viewed as a form of Byzantine failure detection. It is a complementary alternative to consensus voting [8, 28], which is vulnerable to "groupthink" if an attack compromises multiple replicas.

The CATS toolkit embodies the state of the art from research in *authenticated data structures* [2, 6, 12, 20, 21, 24, 25]. As in many of these schemes, CATS servers can supply cryptographic proofs that their read responses resulted from valid write operations. CATS also incorporates new primitives for secure challenges and audits. Servers export a *challenge* interface to verify that writes are incorporated and visible by other clients. The *audit* interface enables an auditor to verify integrity of writes over arbitrary intervals of recent history, so that a faulty server cannot revert writes without risk of detection. Other participants may verify audit results without requiring trust in the auditor. Finally, we extend previous work on authenticated data structures with new capabilities for fast, reliable secondary storage access and caching, and investigate the impact of design choices on storage, access, and accountability costs.

This paper is organized as follows. Section 2 presents an overview of our approach, threat model, and assumptions. Section 3 describes the design of the CATS state management toolkit and storage service. Section 4 outlines the details of our prototype implementation, while Section 5 presents experimental results with the prototype. Section 6 sets our work in context with related work, and Section 7 concludes.

2 Overview

The CATS network storage service is a building block for distributed applications or services. Its purpose is to act as a safe repository of state shared among the participants (actors) of a distributed system, who are clients of the storage service. The shared state guides the behavior of the clients, and the clients may affect the group through updates to the shared state. The clients may represent a community of users, who are responsible for their updates.

CATS supports the core functions of a network file system. Clients can create, read, and write opaque objects identified by unique identifiers (*oids*). Each write request generates a new version of an object. Versions are named by monotonically increasing timestamps of the writes that created them. Reads retrieve the most recent version of the object. A client may also request a prior version of an object that was valid at a specified time. We have limited our focus to essential object storage functions: for example, in its present form, CATS does not support nested directories, symbolic names or links, renaming, partial reads or writes, truncates, or appends.

2.1 Threat Model

The strong accountability property provides clients with the means to verify that the server protects the integrity of the shared state: read responses correctly reflect the valid writes issued by the clients. Moreover, the system can assign provable responsibility for an object version to the client that issued the write for that version.

To define the threat model, we specify the properties of a correctly executing storage service.

- **Authenticity and undeniability.** The service executes writes issued only by properly authorized clients. A client cannot deny responsibility for a write that it has issued.
- **Freshness and consistency.** Writes are applied in order, and reads always return the object version that was valid at the requested time (i.e., the version created by the previous write).
- **Completeness or inclusion.** Writes are included in the service state and are visible to other authorized clients.

A faulty storage server could attempt to violate any of the above properties. For example, it could accept writes from unauthorized clients, improperly modify existing objects, or replay earlier writes. In a more subtle attack the server could acknowledge the completion of a write request but attempt to conceal it from other clients. The storage service or a client could attempt to deny that it executed or requested a completed operation. Table 1 lists possible attacks.

CATS does not prevent any of these forms of misbehavior. In particular, it does not attempt to remove the need for trust among clients and servers; trust is essential for sharing and cooperation. Rather, the philosophy of our approach is: "trust but verify" [35]. CATS holds the storage server and other participants accountable for their actions, so that a faulty actor is exposed and isolated. Accountability precludes an effective attack to subvert the behavior of the overall system without unavoidable risk of detection. Integrity also requires that an accountable system protect its participants from false accusations of guilt. In CATS, a participant's guilt is cryptographically provable to all participants.

CATS is compatible with read access controls. However, violations of confidentiality are not part of the threat model: a CATS server cannot be held accountable for serving reads that violate the policy.

2.2 Trust Assumptions

Table 2 lists the components of a CATS service. The accountability properties of the system rest on correct functioning of two core elements.

Asymmetric cryptography. Each actor can sign its requests using at least one asymmetric key pair bound to a principal; the public key is distributed securely to all actors, e.g., using a Public Key Infrastructure (PKI). Digital signatures ensure integrity, authenticity, and non-repudiation of actions.

Secure authentication is the core trust assumption in our system. If keys are compromised, then an actor may be falsely held accountable for actions it did not take. Note, however, that an attacker cannot misrepresent the actions of any actor whose key it does not possess, and any actions it takes with a stolen key can be traced to that key. Even so, a successful attack against a trusted PKI root would open CATS to subversion by the attacker; thus a PKI constitutes a Trusted Computing Base in the traditional sense.

External publishing medium. Each actor has a means to publish a *digest* of its state periodically. A digest is a secure hash over the actor's state at a point in time. Digests are digitally signed, so an actor cannot repudiate previous claims about its state. Only the storage server publishes digests in the accountable storage service example.

Each actor must have independent access to the history of published digests in order to validate proofs independently. Thus the publishing medium is a trusted

Attack	*Defense*
Server fails to execute write or object create	provable detection
Client repudiates object create or write	provable detection
Server denies write or object create	provable detection
Client writes out of order	rejected by server
Server returns invalid read response	provable detection
Fraudulent or unauthorized writes	provable detection with simple static ACLs; relies on trusted external authorization server for richer policies
Server replays or reverts valid writes, destroys objects, reorders writes, or accepts out-of-order writes	verifiable detection by challenge or audit
Tampering & forking, or other variants of above attacks on data integrity	verifiable detection by challenge or audit
Violation of privacy or read access policy	no defense

Table 1: **Summary of attacks and defenses.**

Component	*Trust Assumptions*
Clients and servers	Trusted but accountable: Cannot subvert the system without risk of provable detection. Incorrect or invalid actions taken with a stolen key are provably detectable.
Publishing medium	Trusted to render published digests visible to all participants. Accountable for forgeries or alterations to published digests. Cannot subvert the system without risk of provable detection.
Authorization service	Not required for simple static access control lists. Must be trusted to enforce richer access control policies if needed.
Trusted platform / trusted path	Necessary for individual user accountability, else no distinction between misbehavior of user and misbehavior of user software.
Public key infrastructure:	Trusted Computing Base: Compromise of PKI can subvert the system.

Table 2: **Summary of components and trust assumptions.**

component. Since digests are signed, a faulty publishing medium cannot forge or alter digests unilaterally, but it could mount a denial-of-accountability attack by concealing them. The medium could also collude with a participant to alter the digest history of that participant, but such an attack would be detectable and provable by another participant that caches previously published digests. In essence, a faulty publishing medium can destroy the accountability properties of the system, but it cannot itself subvert the system.

Due diligence. Accountability in CATS relies on voluntary actions by each actor to verify the behavior of the others. For example, if the clients of the CATS storage service do not request or check proofs that their writes were executed and persist (see 2.3), then a faulty server may escape detection. Of course, a lazy client may free-ride on the diligence of others, possibly leading to a classic tragedy of the commons. What is important is that an attacker cannot determine or control its risk of exposure.

2.3 Challenges and Audits

An important element of our approach is to incorporate *challenge* and *audit* interfaces into service protocols. Challenges force a server to provide a cryptographic proof certifying that its actions are correct and consistent relative to published state digests. An important form of challenge is an *audit* to verify consistent behavior across a sequence of actions or an interval of history. Challenges and audits do not require trust in the auditor; any actor may act as an auditor. If an actor's challenge or audit reveals misbehavior, the actor can present its case as a *plaintiff* to any other actor, which may verify the validity of the accusation.

Auditing defends against a freshness attack, in which a faulty server discards or reverts a valid write that it has previously accepted (Section 3.5). A client with authority to read an object can choose to audit the sequence of updates to that object through time to ensure freshness and consistency. Actors select a degree of auditing that balances overhead and the probability of detection of misbehavior. The server cannot change its history to conceal its misbehavior from an auditor.

For example, a CATS storage server may be challenged to prove that it has incorporated a recently completed write into its published state digest. It may be audited to prove that its current state resulted from a sequence of valid writes by authorized clients, and that its read responses reflect that state. A challenged or audited CATS server must also justify that any accepted write complies with the existing access control policy (See Section 2.4). A faulty server cannot allow unauthorized writes or execute its own writes using a fraudulent identity. If an attacker subverts the server, the worst damage it can cause is to deny service or discard data; covert modifications (including reverted writes) are tamper-evident and can be exposed at any time through challenges and audits. In particular, the server can be held provably accountable for a forking attack [17].

2.4 Discussion and Limitations

User accountability, identity, and privacy. Ideally, the accountability of a system should extend to its users. For example, the CATS storage service could hold users accountable for actions they take within a community or organization, possibly exposing them to legal sanction or other social sanction.

However, user accountability requires strong identity bindings. There is a fundamental tension between accountability and privacy or anonymity. Strong identities exist today within user communities that interact using shared services in areas such as health care, infrastructure control, enterprise work flow, finance, and government, where accountability is particularly important. Many of these areas already use some form of PKI.

User accountability also requires a trusted path to the user so that software running with the user's identity cannot act without the user's consent. The questions of how to establish a trusted path to the user and ensure that the user authorizes actions taken with its identity are outside the scope of this paper. The solution is likely to require some trust in the platform (e. g., a Trusted Platform Module [33]).

More generally, a signed message or action is not a guarantee of intent by the principal: many security breaches occur when components are subverted or private keys are otherwise stolen [3]. Importantly, when such a breach occurs, our approach ensures that attackers cannot modify existing history, as noted above.

External publishing medium. The external publishing medium is a certified log that is globally visible. The publishing medium is much simpler than the storage service because it does not accept updates, does not support dynamic object creation, and does not allow write-sharing. There are several ways to implement an external publishing medium. Timestamping services [7] publish digests using write-once media such as newspapers. An alternative solution may use a trusted web site. Finally, the clients of the service can implement the publishing medium by using a peer-to-peer approach that leverages some form of gossip and secure broadcast.

Access control. As part of its operation a CATS service should demonstrate that a write complies with a write access control policy. It is easy to hold the server accountable for enforcing a simple and static policy for write access control; for example, if the creator of an object provides an immutable list of identities permitted to modify the object. We are investigating how to extend strong accountability to richer access control policies. A CATS server may use an external authorization server to govern access control policy and issue signed assertions endorsing specific clients to write specific objects. However, such an authorization server must be trusted, and it is an open question how to assure accountability for it in the general case.

Denial of service through challenges and audits. One concern is that challenges and audits could be used as the basis for a denial-of-service attack on the server. We view this problem as an issue of performance isolation and quality of service: resources expended to serve audits and challenges may be controlled in the same way as read and write requests. Community standards may define a "statute of limitations" that limits the history subject to audit, and bounds the rate at which each actor may legitimately request audits and challenges.

3 CATS Design

Figure 1 presents a high-level view of the CATS storage service and its components. The service and its clients communicate using the Simple Object Access Protocol (SOAP). Each message is digitally signed. The Public Key Infrastructure (PKI) and the publishing medium constitute the trusted base. Misbehavior of any other actor can be detected and proven.

The storage service has simple and well-defined operations (Table 3), which allows us to design it as a thin layer on top of a generic state store toolkit. The CATS toolkit incorporates common state management functions to organize service state to enable strong accountability. The CATS storage service uses the state store toolkit to ensure the tamper-evidence of its state and to prove to its clients that it maintains its state correctly.

The toolkit's core principle is to separate state management from state transformation due to application logic. The CATS toolkit represents service state as an indexed set of named, typed, elements. The state store accepts updates annotated with a request—digitally signed by the issuer—and links these action records to the updated data objects. The CATS accountable storage service is a simple application of the toolkit because each element corresponds to exactly one storage object, and each update modifies exactly one element. The toolkit provides foundational support for strongly accountable application services, although accountability for more advanced services must be "designed in" to the internal logic, state representations and protocols.

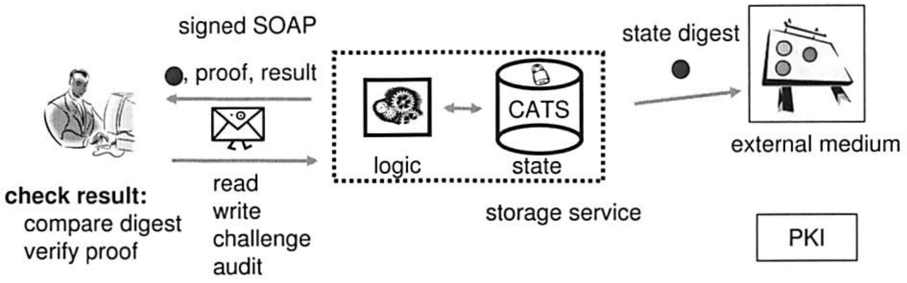


Figure 1: **Overview of a CATS service**, e.g., the accountable storage service.

Operation	*Description*
`read(oid)`	Returns the most recent version of the object with the specified *oid.* In our prototype, every read response includes a proof of correctness relative to the digest for the most recent epoch (implicit challenge).
`read(oid, time)`	Returns a signed response with the version of the specified object at given time (epoch). In our prototype, every read response includes a proof of correctness relative to the digest for the epoch (implicit challenge).
`write(oid, value, current)`	Overwrites the specified object with the given value. The server accepts the request only if at the time of the write the version stamp of the object is equal to *current.* Otherwise, returns the object's version stamp.
`challenge(oid, epoch)`	Requests that the server provide a proof that the object with the given *oid* is or is not included in the service state, relative to the published digest for the given time (epoch). The response includes a membership proof for the object and its value, or an exclusion proof if the *oid* was not valid.
`audit(oid, epoch, span, depth)`	Requests that the service substantiate the write history for the object named by *oid* over the time interval $[epoch, epoch + span]$. The request is equivalent to *depth* randomly selected challenge requests over the interval. The depth parameter controls the degree of assurance.

Table 3: **Accountable storage service operations.** The service accepts reads and writes to a set of named versioned objects. All client writes and all server responses are digitally signed so that actors can be held accountable for their actions. Challenges and audits provide the means for clients or third-party auditors to verify the consistency of a server's actions relative to periodic non-repudiable digests generated by that server and visible to all actors.

3.1 Action Histories

Strong accountability requires CATS to represent action histories whose integrity is protected, and provide primitives to retrieve, exchange, and certify those histories. We have to know who said what to whom, and how that information was used.

Every CATS request and response carries a digital signature that uniquely authenticates its sender and proves the integrity of the message. Specifically, the content of each request and response is encapsulated in a signed *action record.* Actors may retain a history of action records and transmit them to make provable statements about the behavior of other actors and the validity of their actions. The action history may be integrated into the internal service state.

For example, the CATS storage server links each element or object to the set of client requests that affected it or produced its value. A CATS server may justify its responses by presenting a sequence of cached, signed action records that show how its state resulted from the actions of its clients, starting from its published state at a given point in time, as described below.

Action records are verifiable by any receiver and are non-repudiable by the sender. Importantly, symmetric cryptography approaches, e.g., Message Authentication Codes or SSL, are not sufficient for accountability. Shared keys cannot guarantee non-repudiation of origin or content: any party in possession of the shared key can forge the actions of another using the key.

3.2 State Digests and Commitment

A CATS server periodically generates a signed digest over its local state. The state storage data structure provides a function for an actor to compute a compact, fixed-size hash over an atomic *snapshot* of its current state.

Servers publish their digests by including them in their responses to clients and publishing them to an external medium (Section 2.2). Publishing a digest commits the server to a unique view of its state and its history at a specific point in time. The server cannot lie about what its state was at that time without the risk of exposure. Any modification of the state affects the digest; any conflicts with previously published digests are detectable and provable. In particular, the server cannot safely present different views of its state to different clients—a *forking attack* [17, 23]. While nothing prevents the the server from returning different digest values to different clients, clients can detect such misbehavior by comparing digests among themselves or against the published copy.

An *epoch* is the time interval between generation of two consecutive state digests by some actor. An epoch is *committed* when the digest of its end-state is published. Epochs are numbered with consecutive timestamp values. To preserve the integrity of the timeline, the computation of each new state digest incorporates the digest from the previous epoch as an input. This form of chaining prevents the service from removing a committed snapshot or adding a new snapshot in the past.

3.3 Proofs

Servers can make provable statements about their states or operations relative to a published digest. A CATS server can generate two kinds of proofs:

- **Inclusion.** The server claims that a given $(oid, value)$ pair was recorded in the service state at the end of some epoch. The claim is backed by an unforgeable *membership proof* showing that the $(oid, value)$ pair was used to compute the digest for that epoch in a proper way.
- **Exclusion.** The server claims that no object with a given *oid* existed in its state at the end of some epoch. The claim is backed by a proof exhibiting a proper computation of the epoch digest that did not include an object with that *oid*. If a previous writer had written an object with that *oid*, it could have verified that the server incorporated the object into its digest in a proper way that would have precluded such a proof.

Section 3.6 describes these proofs and the data structures that support them in more detail.

A client may request a proof by issuing a *challenge* on a target *oid*. The server responds to a challenge with an inclusion proof if the *oid* was valid, or an exclusion proof if it was not. If the challenge is issued by an actor other than the last writer, an inclusion proof includes a signed copy of the write that generated the object's value.

Challenges enable a client to substantiate any *read* or *write* response. A writer can issue a challenge to verify that its write to an *oid* was incorporated into the server's state for an epoch, as represented in the server's published digest, in a proper way that precludes the server from denying or safely concealing the write at a later time. A reader can issue a challenge to verify that its view of the server's state is consistent with the server's digests, and therefore (by extension) consistent with the views verified by the writers and seen by other readers.

3.4 Request Ordering and Consistency

The state of a stored object results from an ordered sequence of writes to the object. Accountability for object values requires that the server and the object's writers agree on and certify a serial order of the writes.

Each stored object is annotated with a version stamp consisting of a monotonically increasing integer and a hash of the object's value. The action record for an object write must include the object's previous version stamp, certifying that the writer is aware of the object's state at the time of the write. The server rejects the write if the version stamp does not match, e.g., as a result of an intervening write unknown to the writer. The writer may retry its write, possibly after completing a read and adjusting its request.

Inspection of a sequence of write actions to an *oid* will reveal any attempt by the server to reorder writes, or any attempt by a writer to deny its knowledge of an earlier write.

3.5 Freshness and Auditing

It remains to provide a means to hold a server accountable for "freshness": an object's value should reflect its most recent update prior to the target time of the read. A faulty server might accept a write and later revert it. Freshness is a fundamental problem: a server can prove that it does not conceal writes to an object over some interval only by replaying the object's complete write history over the interval, including proofs that its value did not change during any epoch without a valid write action. Reverted writes are the most difficult form of the "forking attack" to defend against; in essence, the *fork consistency* assurance of SUNDR [17, 23] is that if the server conceals writes, then it must do so consistently or its clients will detect a fault.

Our approach in CATS is to provide a means for any participant to selectively audit a server's write histories relative to its non-repudiable state digests. The strong accountability property is that the server cannot conceal any corruption of an object's value from an audit, and any misbehavior detected by an audit is verifiable by a third party. Although reverted writes for objects or intervals that escape audit may go undetected, a server cannot

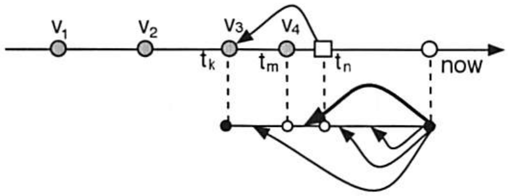


Figure 2: At time t_n the malicious operator reverts the effect of the update performed at time t_m and sets the corresponding value to the one valid at time t_k. An audit that tracks the value through a sequence of snapshots detects this violation, as long as a read is performed in the interval $[t_m; t_n)$.

predict or control what actions will be audited.

Figure 2 illustrates the auditing process. In this example, an object receives an update at time t_m, which is correctly applied and the value of the object becomes v_4. However, at a later time, t_n, the server reverts the value of the element to a previous value v_3, created at time t_k. As a result of this intervention, reads to the object produce an incorrect result (v_3 instead of v_4).

To certify freshness, a client must challenge and examine the object's value for every epoch in the interval since the last reported update to present. We will refer to this interval as the *span* of auditing ((t_k, now) in our example). If the audit confirms that the object and its reported value were incorporated into all digests for the interval, then the client can safely conclude that the reported value is indeed fresh. On the other hand, if the server cannot provide a membership proof for the object and its value relative to some subsequent digest, then the server is faulty.

Clients can reduce the overhead of certifying freshness at the expense of a weaker, probabilistic assurance. Instead of inspecting every snapshot in a span, the client may instead issue challenges for randomly selected snapshots in the span. We refer to the number of audited epochs as the *depth* of the audit. In the above example, the client performs an audit of depth 4. The audit successfully detects the misbehavior, since it inspects the service state in a snapshot created in the interval $[t_m, t_n)$, in which the object has the reverted value v_4.

The CATS toolkit and accountable storage service support probabilistic audits as described in the above example. Clients can issue audit requests and specify a list of randomly selected snapshots from a specified auditing span. CATS then constructs a membership proof for the object and its value in the specified snapshots and returns the result back to the client.

The probability of detecting a reverted update for a given state element depends on the lifetime of the reverted update relative to the span of auditing. Using the above example, as the number of snapshots in $[t_n, now)$ increases, the probability of selecting a snapshot in $[t_m, t_n)$ decreases. Thus, detecting an offense becomes more costly as more time passes since the offense took place. If an incorrect update is not noticed earlier, it may be difficult and expensive to detect it later.

One way to deal with this problem is to examine more snapshots in the span (increase the depth of auditing) when verifying older objects. More formally, if p is the probability of selecting a snapshot that reveals misbehavior, the probability of detecting misbehavior after d snapshot inspections is $P(p, d) = 1-(1-p)^d$. If the lifetime of a reverted update decreases relative to the span, then p decreases. Figure 3 shows the probability that audits of fixed depth detect that the value of an element is incorrect for different values of p and d.

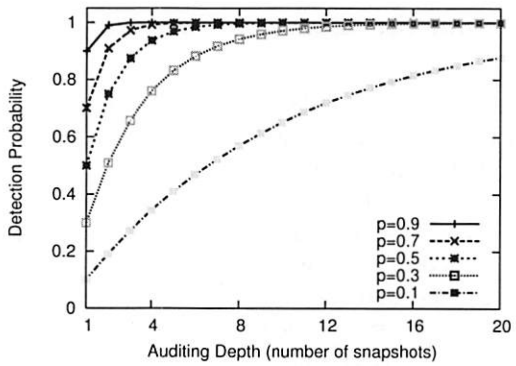


Figure 3: The longer a reverted update remains undetected (decreasing p), audits will have to inspect a larger number of snapshots to ensure a certain probability of detection.

Another possible solution is to have trusted auditors periodically inspect every object and ensure its freshness. The auditors publish commit records visible to all clients, certifying that the service state until the point of inspection is consistent. An alternative, less expensive form of auditing can impose a probabilistic bound on the number of reverted updates. The core idea is to choose objects at random and inspect their values in randomly selected state snapshots. Using a sampling theory result stating that the successful selection of n ($n > 0$) white balls from an urn ensures that with probability $1 - 1/e$ there are no more than an $1/n$ fraction of black balls in the urn, we can prove the following two theorems:

Theorem 1. *Examining the complete execution history of n ($n > 0$) objects chosen uniformly at random from all state objects present in a service's state, and ensuring the correctness of each update operation to these ob-*

jects, guarantees that with probability at least $1 - 1/e$ the service has maintained correctly at least $(1 - 1/n)$ fraction of all objects over its complete execution history.

Theorem 2. *Examining n $(n > 0)$ state update operations and ensuring that none of them reverts a previous update, ensures that with probability at least $1 - 1/e$, no more than an $1/n$ fraction of all state updates have been reverted.*

We can further improve the effectiveness of audits using repetition. It takes $O(\log \delta)$ repetitions of an auditing algorithm to boost the probability of detection to $(1 - \delta)$ for any $\delta \in (0, 1)$.

Finally, another complementary approach is for each writer to challenge periodically objects it has written to ensure that its updates persist until another authorized client overwrites them.

3.6 Data Structure Considerations

We considered two ways to compute a compact digest over a dynamic set of elements and demonstrate that an element with a given *oid* and value is or is not part of the set: cryptographic accumulators [5] and authenticated dictionaries [2, 21, 24]. Of the two, authenticated dictionaries promise to be more practical as they have logarithmic update cost and produce proofs logarithmic in the size of the set (both for inclusion and exclusion). The advantages of authenticated dictionaries prompted us to use them as a basis for the CATS toolkit.

3.6.1 Authenticated Dictionaries

An authenticated dictionary is a map of elements with unique identifiers, and allows for efficient identifier-based updates and retrievals. Most proposed schemes are based on Merkle trees [24]. They organize the elements of the set in a binary search tree. Each node of the tree is augmented with an *authentication label*—the output of a *one-way, collision-resistant* hash function applied on the contents on the node's sub-tree. More specifically, the label of a leaf node is equal to the hash of its data and identifier, and the label of an internal node is equal to the hash of the concatenation of the labels of its left and right children. The recursive scheme labels the root of the tree with an *authenticator* covering the entire tree—its value is a digest computed over every element of the set.

A membership proof for an element constructed by an authenticated dictionary consists of a sequence of sibling hashes: one for each node on the path from the root of the tree to the leaf node containing the element. Using the sibling hashes and the hash of the element's data, an actor can recompute the root authenticator and compare it to the known authenticator. It is computationally infeasible to fabricate a set of sibling labels that combine with the element hash to yield the known root authenticator (e.g., an epoch digest). Exclusion proofs require that the hashes incorporate the key values for each node in the search tree, so that each membership proof also certifies that the path to the object reflects a proper sorted order for the tree. If the tree is in sorted order, then an exclusion proof for a given *oid* is given by the set of hashes for any subtree that covers the portion of the key space containing *oid*, but that does not contain that *oid* itself.

3.6.2 Practical Issues

A number of organizational issues impact the performance of authenticated dictionaries. In this subsection we describe the cost dimensions, the way they impact performance, and the specific choices we made to design the state store.

Tree degree. It is possible to authenticate any search tree simply by allocating extra space within each node to store its authentication label. A naive application of the same idea to a high-degree tree such as a B-Tree [4], however, can have an adverse impact on authentication performance. The high degree used by a B-Tree increases the size of the sibling set needed to compute an authentication label: a membership proof for a B-Tree storing 1,000,000 four-byte identifiers with out-degree of 100 is more than 7 times larger than a proof generated by a binary tree storing the same data.

Theorem 3. *Balanced binary search trees generate membership proofs of minimal size.*

Proof. A balanced tree of degree B storing N objects has height at least $\log_B N$. A membership proof consists of a constant size component for each sibling of every tree node along a root to leaf path. Each tree node has $B - 1$ siblings. Therefore, a membership proof has at least $f(B) = (B - 1) \log_B N$ components. This function obtains its smallest value for $B = 2$. □

The above result exposes a dilemma for building a scalable authenticated state store: indexing large collections requires the use of an I/O efficient data structure, e.g., a B-Tree, however, it will be costly to compute digests and to generate, transmit, and verify proofs in such a data structure. We address this challenge with a hybrid approach consisting of a balanced binary tree mapped to the nodes of a B-Tree for efficient disk storage (Figure 4). While this approach ensures the optimality of search operations, updates can trigger a large number of I/O operations to balance the binary tree. We relax the requirement of global binary tree balance and trade it for improved I/O performance: only the part of the tree that is stored within a single B-Tree block should be balanced. This restriction may increase the height of the binary tree but offers an acceptable tradeoff between membership proof size and I/O efficiency. This solution is similar to the one of Maniatis [21]. Theorem 3 shows that his choice of a binary tree is indeed optimal.

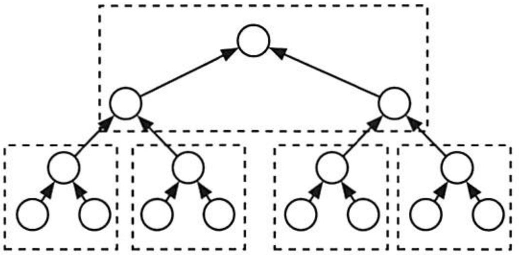


Figure 4: Mapping a binary search tree to disk blocks for efficient storage.

Data layout. Data placement also impacts the performance of authenticated search trees. Search trees provide two options for storing data. Node-oriented organizations store data at all nodes, while leaf-oriented designs store data only at the leaves. These two choices impact the space requirement, size of membership proof, and authentication cost. Leaf-oriented trees require more space but need less data and processing to compute an authentication label [32]. In the context of a binary tree mapped to a B-Tree, the smaller size of internal tree nodes in a leaf-oriented design results in a large portion of the binary tree being mapped to a single B-tree node. This organization reduces the I/O and authentication cost and we adopt it in our design.

Persistence. Since it is expensive to generate, publish, store, and audit epoch digests, it is desirable to minimize the number of epochs for a write stream by maximizing the *epoch length*, the average number of updates within an epoch. While there exist methods for building space-efficient persistent data structures [9] with *constant* space overhead for epoch length of one update, persistent authenticated search trees incur *logarithmic* space overhead; each update invalidates a logarithmic number of authentication labels. Using longer epochs reduces a constant fraction from this overhead by amortizing the cost of tree maintenance (including regenerating the authenticators) across multiple update operations. For example, since labels close to the root of the tree are affected by every update operation, delaying label regeneration to the end of the epoch ensures that each label is computed exactly once. Processing more updates within a single epoch also decreases space utilization and the disk write rate.

However, there are limits to the practical epoch length. Longer epochs include more writes and modify more disk blocks; the number of dirtied blocks for a write set also grows logarithmically with the size of the tree. If the set of dirty blocks does not fit within the I/O cache, then the system will incur an additional I/O cost to regenerate the labels at the end of the epoch. Longer epochs also increase the time to create a state digest and acknowledge the completion of an operation. Epoch length is an adjustable parameter in our design, in order to reflect these complexities.

4 Implementation

We implemented the state store toolkit and the storage service using C#, .NET framework 2.0, and Web Services Enhancements (WSE) 2.0 for SOAP/Direct Internet Message Encapsulation (DIME) communication. The storage service toolkit consists of 12,413 lines of code, and the service implementation took another 6,084 lines. The toolkit implementation consists of several layered modules. The lowest layer exports a block-level abstraction with a unified interface to pluggable underlying storage implementations. The current implementation uses files for storage media so it can easily use parallel or replicated storage supported by the file system.

An intermediary block cache module controls access to storage and buffering. The block cache provides interfaces to pin and unpin blocks in memory. An asynchronous background process monitors the block cache and drains the cache to keep the number of blocks in use between configured high-water and low-water marks. Each block can be locked in shared or exclusive mode. The current implementation does not perform deadlock detection, and it maintains all versions of a given object.

The authenticated dictionary implementation uses the block cache module to access storage blocks. The dictionary is implemented as a B-Tree with the keys within each block organized in a binary search tree. The current implementation uses Red-Black trees and it allows to use other algorithms if necessary. We use Lehman and Yao's B-link tree concurrency algorithm [16] to ensure optimal concurrency. The state store can correctly recover after a failure using write-ahead logging.

The storage service is log-structured [27]. It consists of an append-only log and an index for locating data on the log. We use the toolkit for the index, and a variant of the toolkit's write-ahead log for the append-only log. Each log entry consists of two portions: the original XML request, and the request payload. Clients identify objects using 16-byte key identifiers.

The service is structured as a collection of stages (Figure 5) using a custom staging framework inspired by SEDA [34]. A pool of worker threads services requests for each stage. The size of the pool changes dynamically in response to offered load. Stages have queues of fixed capacity to absorb variations in load. Once queues fill

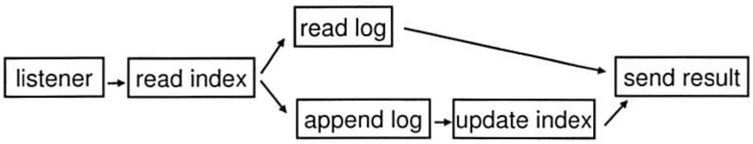


Figure 5: **Storage service implementation.** The storage service consists of an index and an append-only log. The service is implemented as a collection of stages. Each stages is associated with a pool of worker threads. Pools can grow and shrink depending on load.

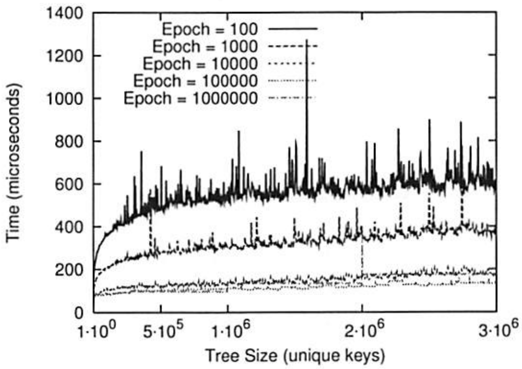


Figure 6: The time to apply an update to the authenticated dictionary is logarithmic relative to the number of unique keys in the index and increases as epoch length decreases.

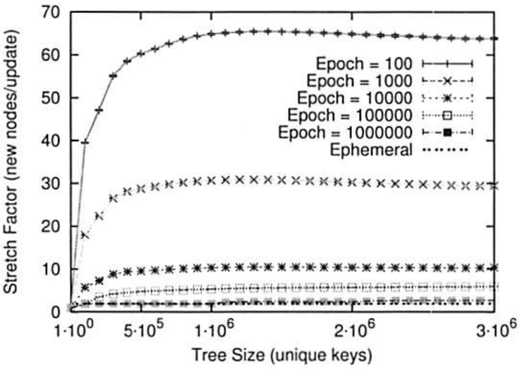


Figure 7: The average number of new nodes introduced per update operation, *stretch factor*, is logarithmic relative to the epoch length. Smaller epochs have significantly higher stretch factors resulting in larger indexes.

up, the service starts rejecting requests. Clients resubmit rejected requests at a later time.

The service and its clients communicate via SOAP over TCP. To limit the overhead of XML, we transfer binary data using DIME attachments. Requests and responses are digitally signed. The current implementation uses a custom XML canonicalization scheme and RSA for signatures. Future releases will integrate the service with web services standards such as XML-Signatures and WS-Security.

The service processes update requests in epochs. The service creates a new epoch as soon as it receives a valid write request for an object that has already been modified in the current epoch. This approach allows for the best granularity of accountability as it preserves every version of an object. However, it has impact on performance if workload has high contention (Section 5.4). An alternative implementation choice is to limit the rate at which new epochs are created by imposing a minimum bound on the length of an epoch. To enforce this bound, the service must reject a write if it will generate an epoch ahead of its due time. Clients must buffer and coalesce their writes, which will result in decreased data sharing.

5 Evaluation

In this section we present an evaluation study of our prototype. Our goal is to understand how accountability impacts performance under varying conditions. In particular, we study the impact of public key cryptography, object size, request rate, workload contention, epoch length, and auditing.

5.1 Methodology

We run all tests on IBM x335 servers running Windows 2003 Server Standard Edition operating system. Each machine has 2 Pentium IV Xeon processors running at 2.8GHz, 2GB of RAM, 1GB Ethernet, and IBM SCSI hard disks with rotation speed of 10,000 RPM, average latency and seek time of 3 ms and 4.7 ms respectively.

We use a population of 1,000,000 unique objects and pre-populate the server to have one version of each object. Since the cost of index operations grows logarithmically with the size of the state store, a state store with 1,000,000 objects has already reached a steady state in which increasing size has minor impact on performance (doubling the size of the store will increase the cost of operations by approximately 3%).

We use a community of clients to issue requests to the service using the SOAP interface. Our workloads consist of synthetic reads/writes of a controlled size. We vary

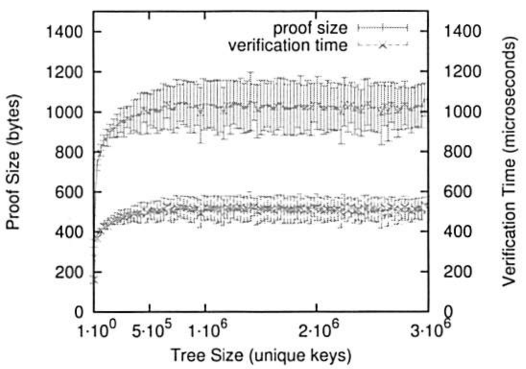


Figure 8: Membership proof size is logarithmic relative to the number of unique keys in the index and is on the order of 1KB. The time to verify a membership proof is on the order of 550 μseconds.

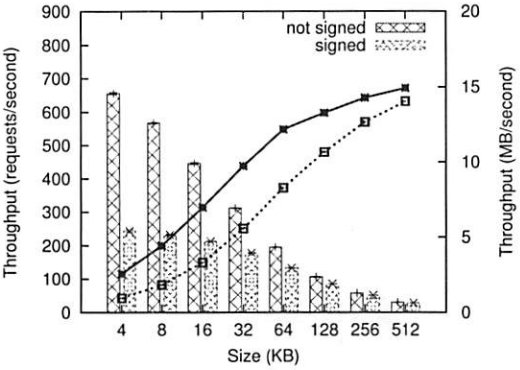


Figure 9: Maximum achievable data bandwidth for storage service RPC interface based on SOAP/DIME and WSE 2.0. The results suggest that this interface has high overhead and services using it will be potentially communication-bound.

the "heat" of the workload by biasing the probability of selecting a given object as the target of a read/write operation. We denote heat using the notation $X : Y$, which we interpret to mean: $X\%$ of the requests go to $Y\%$ of the objects. For example 100:100 is a uniform workload, and 80:20 is the typical hot/cold workload. All tests last 3 minutes with initial 30 seconds for warming the caches. We report averages and standard deviations from 10 runs. Our test workloads are a rough approximation of real storage workload and the magnitude of the performance results we report may differ from that of real systems. However, the focus of our study is the cost of accountability and our workloads allow us to vary the key parameters to quantify those effects.

We use SHA-1 to compute authentication labels. Object keys are assigned randomly in the identifier space (16 bytes). This is a conservative approach intended to limit spatial locality and stress our implementation. In practice, spatial locality can reduce the overhead of persistence and the cost of state digest computation. We store the index and the log in blocks of size 64KB. The index cache has capacity 2000 blocks and the log cache has capacity 8000 blocks. The cache sizes are chosen so that neither the index, nor the append-only log fit in memory. For storage medium we use files residing on the local NTFS file system. The files are accessed using the standard .NET I/O API. For improved performance we place the append-only log and the index on separate disks. We commit dirty blocks to disk every 2 seconds.

5.2 Toolkit Behavior

We first examine the behavior of the CATS toolkit. In particular, we study membership proofs (size and time to verify), and the impact of epoch length on update and storage costs. We use a set of 3,000,000 random keys of size 16 bytes and insert them consecutively in a CATS index. We create new epochs at a controlled rate.

Figure 6 shows the time to apply an update to the authenticated index as a function of the unique keys in the index and epoch length. We observe that the update cost is logarithmic with respect to the number of unique keys. The cost also grows as epoch length decreases. There are two primary reasons to explain this behavior. First, smaller epochs amortize the cost of maintaining the index and computing authentication labels among a smaller number of updates. Second, smaller epochs produce larger indexes and cause higher write rate per update operation.

Next we examine the impact of epoch length on storage costs. We define the *stretch factor* for a given epoch length to be the average number of new (Red-Black tree) nodes introduces by an update operation. Ephemeral trees have epoch length of infinity and stretch factor of 2. Figure 7 shows how the stretch factor changes with epoch length. The stretch factor is logarithmic relative to the number of unique objects in the index and grows as epoch length decreases. Thus very small epochs can produce very large indexes.

Finally, we examine membership proofs. Figure 8 shows the size of membership proofs and the time to verify a membership proof as a function of the number of unique keys in the index. Importantly, these metrics do not depend on epoch size. As expected, membership proof size is logarithmic relative to the unique keys in the index. For our configuration, a proof is on the order of 1KB and it takes about 550 μseconds to verify.

The important point to take away from these experiments is that membership proofs are compact and relatively cheap to verify. Update operations in our unoptimized prototype have acceptable performance. Epoch

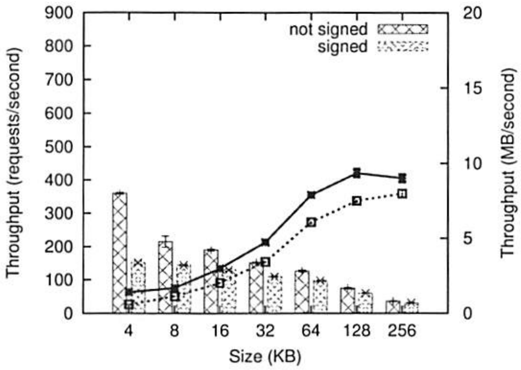


Figure 10: Write saturation throughput for objects of different size using 80:20 workload for configurations with and without digital signatures. As object size increases, the relative cost of digital signatures decreases.

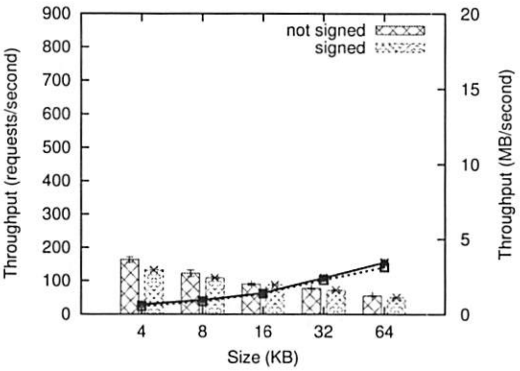


Figure 11: Read saturation throughput for objects of different size using 80:20 workload for configurations with and without digital signatures. Performance is seek limited due to the log-structured design.

length has direct impact on space overhead and update cost and is the primary factor that determines performance.

5.3 Saturation Throughput

We now measure the saturation throughput of the storage service prototype. The first experiment evaluates the maximum bandwidth a storage service using SOAP/DIME based on WSE 2.0 can achieve. We issue write requests to the service to store objects of controlled sizes. The client follows the full protocol, while the server only extracts the object from the SOAP message, validates it (when requests are signed) and sends a response to acknowledge the operation. No storage takes place. The server's CPU is saturated. Figure 9 shows the saturation request rate and the resulting data bandwidth for objects of different size with and without using digital signatures. Although we are using 1Gbps LAN, the observed data bandwidth peeks at approximately 15MB/s. This is the best possible request rate that our storage service can achieve. Importantly, the results show that digital signatures are expensive, however, the relative cost signing a request decreases as object size increases.

In the next experiments we measure the read and write saturation throughput of our implementation under a traditional hot/cold workload (80:20). We use the load generator to fully saturate the server and measure the sustained throughput. We vary object size to study its impact. Figure 10 shows the performance for write and Figure 11 for read operations. As object size increases, the request rate decreases and the transfer rate increases. With the increase of object size, write performance increases relatively to the maximum achievable request rate. The service offers better write than read performance, which is due to its log-structured design.

As object size increases the relative penalty of using digitally signed communication decreases. 4KB signed writes achieve 42% of the request rate of writes without signatures. This number increases to 86% for object of size 256KB. The ratios are better for reads: 81% for 4KB, and 90% for 64KB. The log-structured design explains this observation: the read workload is seek-dominated and the time spent signing a request is smaller relative to the time spent obtaining the data from the storage medium.

Overall, the experimental results show that our implementation offers reasonable performance for both read and write operations and that the cost of digital signatures is not prohibitive.

5.4 Workload Contention

In Section 5.2 we concluded that epoch length has significant impact on performance. The storage service creates a new epoch if it receives more than one update operation to the same object in the same epoch. As a result of this design choice, the performance of the service will be dependent on the contention present in its workload. In the next experiments we study this relationship.

We use workloads of different heat, ranging from uniform (100:100) to highly skewed (80:1) and generate requests at controlled rates bellow the server saturation level. Figure 12 shows the observed epoch creation rate. For a given workload heat, processing requests at a higher rate creates new epochs more frequently. As a result, well-provisioned services processing high volumes of client requests will produce a large number of state snapshots. Similarly, more skewed workloads create new epochs at a higher rate. Clearly, mechanisms are needed to allow services to discard state snapshots to

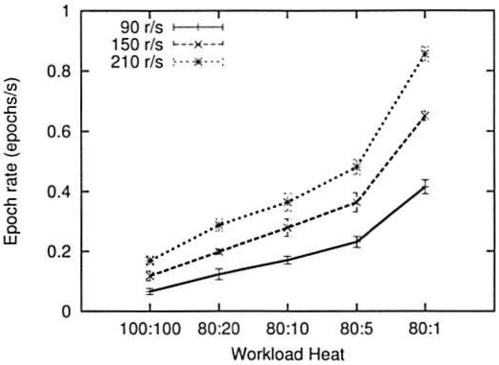


Figure 12: Rate of epoch creation as a function of request rate and workload contention. Higher request rates and contention create new epochs faster and can affect service longevity.

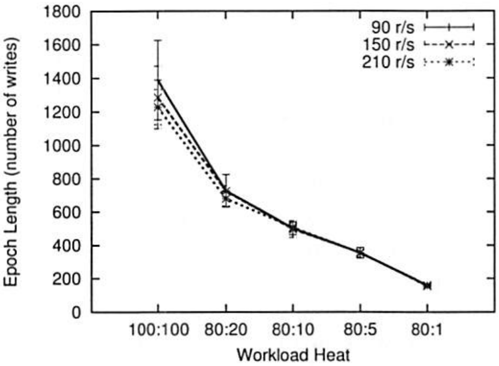


Figure 13: Epoch length as a function of request rate and workload contention. Epoch length is independent on request rate and is determined by workload contention.

keep space overhead and auditing costs at an acceptable level. Dealing with this problem is part of our future work. Figure 13 shows the resulting epoch lengths. Importantly, epoch length does not depend on request rate, however, it can vary from 1400 updates for a uniform workload, to 200 updates for a highly skewed workload. These results suggest that for typical usage scenarios index update operations (the primary cost component of accountability) will be in the range from 300 to 600 μseconds (Figure 6).

5.5 Challenges and Audits

Challenges and audits are the primary mechanisms to ensure a service behaves correctly over time. In the next experiments we study the impact of these mechanisms on service performance. The metric of concern is saturation throughput. For these experiments we populate the service so that the 1,000,000 objects are distributed among 100 epochs of length 10,000 updates. The resulting index is approximately 15 times larger than the index with all objects stored in an epoch of length 1,000,000. To isolate the impact of storage access on auditing, communication is not signed.

The cost of auditing operations depends on their *depth*, the number of inspected snapshots, *span*, the total number of snapshots in the examined interval, and *scope*, the fraction of objects that are likely to be audited, e.g., in our framework objects are audited only on access, and stale objects may never be audited. Using this terminology, a challenge is an audit of depth 1 and span 1.

Figure 14 shows the auditing saturation throughput for different combinations of span, scope, and depth. For a fixed scope, auditing becomes more expensive as span increases. Similarly, higher auditing depths are more expensive. Larger scope is also more expensive, however, its impact is less pronounced. We can explain the above results with the fact that the span and scope determine the locality of auditing operations: smaller spans and scopes concentrate audits on a portion of the index. Increasing the depth queries more snapshots and reduces the total audits/seconds rate.

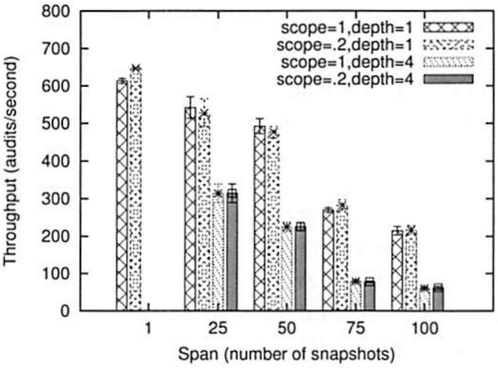


Figure 14: Auditing saturation throughput for different depths, spans, and scopes. Smaller spans show better performance due to increased locality. The impact of scope is less pronounced.

Controlling the rate of auditing is an important problem as it can be used to mount denial of service attacks. This is a resource control problem and solutions from the resource management domains are applicable. For example, clients can be allocated auditing budgets and the service can reject audits from a client if the client has exhausted her budget. Importantly, the services can be challenged for such rejections, to which it can reply with a collection of signed client requests to show that the client has exhausted its budget.

6 Related Work

Butler Lampson summarizes accountability as follows: "Accountability means that you can punish misbehavior". Our community often uses the term to refer to a means of enforcing an authorization policy by detecting and punishing violations after the fact, as an alternative to incurring the cost of preventing them in the first place [14]. More recent work seeks to enforce accountability for performance behavior as essential to effective functioning of networking markets [15].

Our notion of "semantic" accountability is different and complementary. We consider actors accountable if they can demonstrate that their actions are *semantically* correct as well as being properly authorized. An accountable server can be held responsible if its responses violate semantic expectations for the service that it provides. Accountability is "strong" if evidence of misbehavior is independently verifiable by a third party. Clients of a strongly accountable server are also strongly accountable for their actions, since they cannot deny their actions or blame the server.

Designing networked systems for accountability complements classical approaches to building trustworthy systems such as Byzantine Fault Tolerance [8, 28], secure hardware [31], and secure perimeters [14].

Secure perimeters attempt to *prevent* misbehavior by means of authorization and authentication. However, when actors reside in different administrative domains, there is no common perimeter to defend. Once the integrity of the perimeter is violated, it offers no protection, and even authorized actors may abuse the system to their own advantage.

Byzantine fault tolerance (BFT) is a general and powerful technique to *tolerate* misbehavior through replication and voting. However, BFT is vulnerable to collusion: a majority of replicas may collude to "frame" another. Also, in its pure form, BFT assumes that failures are independent, and thus it is vulnerable to "groupthink" in which an attack succeeds against a quorum of replicas. BFT offers limited protection when a service (and hence its replica set) is itself acting on behalf of an entity that controls its behavior (typical of web services). Strongly accountable systems can produce proofs of misbehavior that do not depend on voting or consensus, which makes statements of correctness or misbehavior provable to an external actor.

Secure hardware offers foundational mechanisms to ensure untampered execution at the hardware level. While a secure hardware platform protects the integrity of a programs code, it cannot ensure that its actions are semantically correct. Secure hardware can help provide a trusted path to the user by avoiding corruption of the software stack trusted by the user to correctly represent the user's intent.

Secure logging and auditing can also be used to preserve a tamper-evident record of events [29, 30]. In addition to maintaining a tamper-evident record of service events, we address the question of representing service state to make state management operations verifiable.

CATS embodies the state of the art from research in *authenticated data structures* [2, 6, 21, 25], beginning with the well-known technique of Merkle trees [24]. Relative to that work, it incorporates new primitives for secure auditing and integrates concurrency and recovery techniques from the database community. Our approach is inspired by Maniatis [21].

Our work incorporates these techniques into a toolkit that can act as a substrate for a range of accountable services. The elements of CATS have been used in many previous systems for similar goals of semantic accountability. Many systems require digitally signed communication and some maintain some form of signed action histories [11, 21]. Long-term historic state trails help establish provable causality in distributed systems [22]. Storage systems often reference data using cryptographic hashes to ensure tamper-evidence [26]. State digests are used to prove authenticity in file systems [10], applications running on untrusted environments [19], and time-stamping services [7].

The CATS toolkit is based on the storage abstraction of a binary search tree. The Boxwood Project [18] also provides a toolkit for building storage infrastructures from foundational search tree abstractions. Boxwood does not address the problem of accountability.

SUNDR [17, 23] and Plutus [13] are two recent network storage systems that defend against attacks mounted by a faulty storage server. These services are safe in the sense that clients may protect data from the server, and the clients can detect if the server modifies or misrepresents the data. Plutus emphasizes efficient support for encrypted sharing, while SUNDR defends against attacks on data integrity, the most difficult of which is a "forking attack". In both Plutus and SUNDR file system logic is implemented by clients: the server only stores opaque blocks of data. In SUNDR the server also participates in the consistency protocol.

Both SUNDR and Plutus can detect various attempts by the server to tamper with the contents of stored blocks. However, the papers do not attempt to define precise accountability properties. In general, they blame any inconsistencies on the server: it is not clear to what extent clients are accountable for their actions or for managing file system metadata correctly. For example, SUNDR does not show how to defend against a client that covertly deletes a file created by another user in the same group.

We argue for a stronger notion of accountability in which the guilt or innocence of the server *or its clients* is provable to a third party. This is a subtle change of focus as compared to SUNDR, where the primary concern is to build a tamper-evident system and terminate its use once misbehavior is detected. CATS file semantics are implemented and enforced by the server in the conventional way, rather than implemented entirely by the clients. Clients can independently verify correct execution relative to published digests, rather than "comparing notes" to detect inconsistencies. CATS uses similar techniques to SUNDR and Plutus to make the stored data tamper-evident, but it also defends against false accusations against a server , e.g., a malicious client that "frames" a server by claiming falsely that it accepted writes and later reverted them. Accountability extends to the clients: for example, if a client corrupts shared data, the server can identify the client and prove its guilt even after accepting writes to other parts of the tree.

The BAR model [1] addresses the problem of building cooperative peer-to-peer services across multiple administrative domains. This is a Byzantine Fault Tolerance approach optimized for the presence of rational peers. Since the underlying primitives depend on voting and replication, BAR-based services are vulnerable to collusion. However, BAR does not maintain history and does not use auditing to deal with problems that may arise if peers collude. Our approach shares similar goals with BAR and is complementary. We address the problem of building an accountable service, within the control of a single authority, that is accessed by multiple clients, who may reside in different administrative domains. In this context, the service implements all logic, and voting and replication are not sufficient. Our methodology forces the service to preserve a faithful record of its history to enable auditing in the future. BAR-based services may be able to enforce performance guarantees, something we currently do not support, although accusations are not provable outside the system.

7 Summary and Conclusions

This paper presents the design, implementation, and evaluation of CATS—a storage service and state storage toolkit with strong accountability properties. A distributed system is strongly accountable when participants have the means to verify semantically correct operation of other participants, and assertions or proofs of misbehavior are independently verifiable by a third party based on cryptographically assured knowledge, without reliance on voting or consensus.

CATS builds on a history of work on authenticated data structures, and incorporates the state of the art in that area into a prototype network storage service. Challenge and audit interfaces allow clients, peers, or auditors to verify the integrity of the storage service state through time, providing probabilistic defense against reverted or replayed writes or replayed requests and other attacks on the integrity of the data or the write stream. The probability of detection of a reverted state update is a function of the depth and rate of auditing.

Experimental results with the prototype show that the CATS approach to strong accountability has potential for practical use in mission-critical distributed services with strong identity. The cost of authenticated communication is not prohibitive and its relative overhead decreases with the size of the object set and the stored objects. Workload write contention and write-sharing are the primary factors that affect the cost of accountable storage in our prototype: contention and sharing increase the storage, computation, and auditing costs. The cost of auditing depends on object age and the desired level of assurance.

The accountable storage service described in this paper is a simple example of a strongly accountable service built using the CATS state storage toolkit. It illustrates a state-based approach to building accountable network services that separates service logic from internal state representation and management, associates state elements with the digitally signed requests of the clients responsible for them, and supports verifiably faithful execution of state management operations. In future work we plan to explore use of the state-based approach to build more advanced file services and application services with strong accountability.

Acknowledgments

We thank the reviewers for their helpful comments, and especially our shepherd, Erik Riedel. We also thank Petros Maniatis for his comments and assistance in understanding some subtleties of previous work on authenticated data structures, including his own. Partial support for this research was provided by Duke University and by related funding from IBM and Network Appliance. This research was not supported by the National Science Foundation.

References

[1] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR Fault Tolerance for Cooperative Services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[2] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. In *4th International Conference on Information Security*, October 2001.

[3] R. J. Anderson. Why Cryptosystems Fail. *Communications of the Association for Computing Machinery*, 37(11):32–40, November 1994.

[4] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972.

[5] J. Benaloh and M. de Mare. One-way Accumulators: A Decentralized Alternative to Digital Signatures (Extended Abstract). In *Proceedings of Advances on Cryptology*, pages 480–494, May 1997.

[6] A. Buldas, P. Laud, and H. Lipmaa. Accountable Certificate Management Using Undeniable Attestations. In *Proceedings of the 7th ACM Conference of Computer and Communications Security*, pages 9–17, November 2000.

[7] A. Buldas, P. Laud, H. Lipmaa, and J. Villemson. Timestamping with Binary Linking Schemes. In *Proceedings of Advances on Cryptology*, August 1998.

[8] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

[9] J. Driscoll, N. Sarnak, D. D. Sleator, and R. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[10] K. Fu, M. F. Kaashoek, and D. Maziéres. Fast and Secure Distributed Read-only File System. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 181–196, October 2000.

[11] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, pages 133–148, October 2003.

[12] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing. In *Proceedings of DARPA Information Survivability Conference and Exposition*, pages 68–82, January 2001.

[13] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies*, March 2003.

[14] B. W. Lampson. Computer Security in the Real World. In *Proceedings of the Annual Computer Security Applications Conference*, December 2000.

[15] P. Laskowski and J. Chuang. Network Monitors and Contracting Systems: Competition and Innovation. In *Proceedings of SIGCOMM*, September 2006.

[16] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems*, 6:650–670, December 1981.

[17] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 91–106, December 2004.

[18] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 105–120, December 2004.

[19] U. Maheshwari, R. Vingralek, and W. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, October 2000.

[20] P. Maniatis. *Historic Integrity in Distributed Systems*. PhD thesis, Stanford University, August 2003.

[21] P. Maniatis and M. Baker. Enabling the Archival Storage of Signed Documents. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 31–45, January 2002.

[22] P. Maniatis and M. Baker. Secure History Preservation through Timeline Entanglement. In *Proceedings of the 11th USENIX Security Sysmposium*, August 2002.

[23] D. Mazières and D. Shasha. Building Secure File Systems out of Byzantine Storage. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.

[24] R. C. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 Sysmposium on Security and Privacy*, pages 122–133, April 1980.

[25] M. Naor and K. Nissim. Certificate Revocation and Certificate Update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000.

[26] S. Quinlan and S. Dorward. Venti: a New Approach to Archival Storage. In *First USENIX conference on File and Storage Technologies*, January 2002.

[27] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, July 1991.

[28] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[29] B. Schneier and J. Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In *Proceedings of the 7th USENIX Security Symposium*, pages 53–62, January 1998.

[30] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, May 1999.

[31] S. W. Smith, E. R. Palmer, and S. Weingart. Using a High-Performance, Programmable Secure Coprocessor. In *Financial Cryptography*, pages 73–89, February 1998.

[32] R. Tamassia and N. Triandopoulos. On the Cost of Authenticated Data Structures, 2003.

[33] Trusted Computing Group. Trusted platform module specification. `https://www.trustedcomputinggroup.org/groups/tpm/`.

[34] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[35] A. Yumerefendi and J. S. Chase. Trust but Verify: Accountability for Network Services. In *Proceedings of the 11th ACM SIGOPS European Workshop*, September 2004.

Design and Implementation of Verifiable Audit Trails for a Versioning File System[†]

Zachary N. J. Peterson
Johns Hopkins University

Randal Burns
Johns Hopkins University

Giuseppe Ateniese
Johns Hopkins University

Stephen Bono
Johns Hopkins University

Abstract

We present constructs that create, manage, and verify digital audit trails for versioning file systems. Based upon a small amount of data published to a third party, a file system commits to a version history. At a later date, an auditor uses the published data to verify the contents of the file system at any point in time. Digital audit trails create an analog of the paper audit process for file data, helping to meet the requirements of electronic records legislation. Our techniques address the I/O and computational efficiency of generating and verifying audit trails, the aggregation of audit information in directory hierarchies, and independence to file system architectures.

1 Introduction

The advent of Sarbanes-Oxley (SOX) [40] has irrevocably changed the audit process. SOX mandates the retention of corporate records and audit information. It also requires processes and systems for the verification of the same. Essentially, it demands that auditors and companies present proof of compliance. SOX also specifies that auditors are responsible for the accuracy of the information on which they report. Auditors are taking measures to ensure the veracity of the content of their audit. For example, KPMG employs forensic specialists to investigate the management of information by their clients.

Both auditors and companies require strong audit trails on electronic records: for both parties to prove compliance and for auditors to ensure the accuracy of the information on which they report. The provisions of SOX apply equally to digital systems as they do to paper records. By a "strong" audit trail, we mean a verifiable, persistent record of how and when data have changed.

Current systems for compliance with electronic records legislation meet the record retention and metadata requirements for audit trails, but cannot be used for verification. Technologies such as continuous versioning file systems [14, 22, 33, 27, 38] and provenance-aware storage systems [26] may be employed in order to construct and query a data history. All changes to data are recorded and the system provides access to the records through time-oriented file system interfaces [31]. However, for verification, past versions of data must be immutable. While such systems may prevent writes to past versions by policy, histories may be changed undetectably (see Section 3).

The digital audit parallels paper audits in process and incentives. The digital audit is a formal assessment of an organization's compliance with legislation. Specifically, verifying that companies retain data for a mandated period. The audit process does not ensure the accuracy of the data itself, nor does it prevent data destruction. It verifies that data have been retained, have not been modified, and are accessible within the file system. To fail a digital audit does not prove wrongdoing. Despite its limitations, the audit process has proven itself in the paper world and offers the same benefits for electronic records. The penalties for failing an audit include fines, imprisonment, and civil liability, as specified by the legislation.

We present a design and implementation of a system for verification of version histories in file systems based on generating message authentication codes (MACs) for versions and archiving them with a third party. A file system commits to a version history when it presents a MAC to the third party. At a later time, a version history may be verified by an auditor. The file system is challenged to produce data that matches the MAC, ensuring that the system's past data have not been altered. Participating in the audit process should reveal nothing about the contents of data. Thus, we consider audit models in which organizations maintain private file systems and publish privacy-preserving, one-way functions of file data to third parties. Published data may even be stored publicly, *e.g.* on a Web page.

[†] This is the fully developed version of a work-in-progress paper that appeared as a *short paper* at the 2005 ACM StorageSS Workshop [7].

Our design goals include minimizing the network, computational, and storage resources used in the publication of data and the audit process. I/O efficiency is the central challenge. We provide techniques that minimize disk I/O when generating audit trails and greatly reduce I/O when verifying past data, when compared with adapting a hierarchy of MACs to versioning systems [13]. We employ incremental message authentication codes [4, 5, 6] that allow MACs to be computed based only on data that have changed from the previous version. Incremental MAC generation uses only data written in the cache, avoiding read I/O to file blocks on disk. Sequences of versions may be verified by computing a MAC for one version and incrementally updating the MAC for each additional version, performing the minimum amount of I/O. Our protocol also reduces network I/O. With incremental computation, a natural trade-off exists between the amount of data published and the efficiency of audits. Data may be published less frequently or on file system aggregates (from blocks into files, files into directories, etc.) at the expense of verifying more data during an audit.

Our solution is based on keyed, cryptographic hash functions, such as HMAC-SHA1 [3]. Public-key methods for authenticating data exist [28] and provide unique advantages over symmetric-key solutions. For instance, during an audit, a file system would reveal its public key to the auditor, allowing the auditor to verify data authenticity only. The auditor would not have the ability to create new, authentic records. With symmetric-key hash functions, when the key is revealed to the auditor, the auditor could also create authentic records, leaving open the possibility of falsifying data. This is out of the scope of our attack model. The auditor is a trusted and independent entity. In this paper, we do not consider a public-key implementation, because public-key operations are far too costly to be used in practice.

Our techniques are largely file system independent in that they do not require a specific metadata architecture. This allows verifiable audit trails to be implemented on a wide variety of systems. Additionally, our design makes the audit robust to disk failures, immune to backup and restore techniques, and allows for easy integration into information life-cycle management (ILM) systems.

We have implemented authentication using incremental MACs in the ext3cow file system. Ext3cow is a freely-available, open-source file system designed for version management in the regulatory environment [31]. Experimental results show that incremental MACs increase performance by 94% under common workloads when compared with traditional, serial hash MACs.

2 Related Work

Most closely related to this work is the SFS-RO system [13], which provides authenticity and integrity guarantees for a read-only file system. We follow their model for both the publication of authentication metadata, replicated to storage servers, and use similar hierarchical structures. SFS-RO focuses on reliable and verifiable content distribution. It does not address writes, multiple versions, or efficient constructs for generating MACs.

Recently, there has been some focus on adding integrity and authenticity to storage systems. Oceanstore creates a tree of secure hashes against the fragments of an erasure-coded, distributed block. This detects corruption without relying on error correction and provides authenticity [42]. Patil *et al.* [30] provide a transparent integrity checking service in a stackable file system. The interposed layer constructs and verifies secure checksums on data coming to and from the file system. Haubert *et al.* [15] provide a survey of tamper-resistant storage techniques and identify security challenges and technology gaps for multimedia storage systems.

Schneier and Kelsey describe a system for securing logs on untrusted machines [37]. It prevents an attacker from reading past log entries and makes the log impossible to corrupt without detection. They employ a similar "audit model" that focuses on the detection of attacks, rather than prevention. As in our system, future attacks are deterred by legal or financial consequences. While logs are similar to version histories, in that they describe a sequence of changes, the methods in Schneier and Kelsey secure the entire log, *i.e.* all changes to date. They do not authenticate individual changes (versions) separately.

Efforts at cryptographic file systems and disk encryption are orthogonal to audit trails. Such technologies provide for the privacy of data and authenticate data coming from the disk. However, the guarantees they provide do not extend to a third party and, thus, are not suitable for audit.

3 Secure Digital Audits

A digital audit of a versioning file system is the verification of its contents at a specific time in the past. The audit is a challenge-response protocol between an auditor and the file system to be audited. To prepare for a future audit, a file system generates authentication metadata that commits the file system to its present content. This metadata are published to a third party. To conduct an audit, the auditor accesses the metadata from the third party and then challenges the file system to produce information consistent with that metadata. Using the security constructs we present, passing an audit establishes that the file system has preserved the exact data used to generate authentication metadata in the past. The audit process applies to individual files, sequences of versions, directory hierarchies, and an entire file system.

Our general approach resembles that of digital signature and secure time-stamp services, *e.g.* the IETF Time-Stamp Protocol [1]. From a model standpoint, audit trails extend such services to apply to aggregates, containers of multiple files, and to version histories. Such services provide a good example of systems that minimize data transfer and storage for authentication metadata and reveal nothing about the content of data prior to audit. We build our system around message authentication codes, rather than digital signatures, for computational efficiency.

The publishing process requires long-term storage of authenticating metadata with "fidelity"; the security of the system depends on storing and returning the same values. This may be achieved with a trusted third party, similar to a certificate authority. It may also be accomplished via publishing to censorship-resistant stores [41].

The principal attack against which our system defends is the creation of false version histories that pass the audit process. This class of attack includes the creation of false versions – file data that matches published metadata, but differ from the data used in its creation. It also includes the creation of false histories; inserting or deleting versions into a sequence without detection.

In our audit model, the attacker has complete access to the file system. This includes the ability to modify the contents of the disk arbitrarily. This threat is realistic. Disk drives may be accessed directly through the device interface and on-disk structures are easily examined and modified [12]. In fact, we feel that the most likely attacker is the owner of the file system. For example, a corporation may be motivated to alter or destroy data after it comes under suspicions of malfeasance. The shredding of Enron audit documents at Arthur Anderson in 2001 provides a notable paper analog. Similarly, a hospital or private medical practice might attempt to amend or delete a patient's medical records to hide evidence of malpractice. Such records must be retained in accordance with HIPAA [39].

Obvious methods for securing the file system without a third party are not promising. Disk encryption provides no benefit, because the attacker has access to encryption keys. It is useless to have the file system prevent writes by policy, because the attacker may modify file system code. Write-once, read-many (WORM) stores are alone insufficient, as data may be modified and written to a new WORM device.

Tamper-proof storage devices are a promising technology for the creation of immutable version histories [24]. However, they do not obviate the need for external audit trails, which establish the existence of changed data with a third party. Tamper-resistant storage complements audit trails in that it protects data from destruction or modification, which helps prevent audit failures after committing to a version history.

4 A Secure Version History

The basic construct underlying digital audit trails is a message authentication code (MAC) that authenticates the data of a file version and binds that version to previous versions of the file. We call this a *version authenticator* and compute it on version v_i as

$$A_{v_i} = \mathrm{MAC}_K(v_i || A_{v_{i-1}}); A_{v_0} = \mathrm{MAC}_K(v_0 || N) \quad (1)$$

in which K is an authentication key and N is a nonce. N is a randomly derived value that differentiates the authenticators for files that contain the same data, including empty files. We also require that the MAC function reveals nothing about the content of the data. Typical MAC constructions provide this property. CBC-MAC [2, 16] and HMAC-SHA1 [3] suffice.

By including the version data in the MAC, it authenticates the content of the present version. By including the previous version authenticator, we bind A_{v_i} to a unique version history. This creates a keyed hash chain and couples past versions to the current authenticator. The wide application of one-way hash chains in password authentication [20], micropayments [34], and certificate revocation [23] testifies to their utility and security.

The authentication key K binds each MAC to a specific identity and audit scope. K is a secret key that is selected by the auditor. This ensures keys are properly formed and meet the security requirements of the system. During an audit, the auditor verifies all version histories authenticated with K. Keys may be generated to bind a version history to an identity. A file system may use many keys to limit the scope of an audit, *e.g.* to a specific user. For example, Plutus supports a unique key for each authentication context [17], called a *filegroup*. Authentication keys derived from filegroup keys would allow each filegroup to be audited independently.

A file system commits to a version history by transmitting and storing version authenticators at a third party. The system relies on the third party to store them persistently and reproduce them accurately, *i.e.* return the stored value keyed by file identifier and version number. It also associates each stored version authenticator with a secure time-stamp [21]. An audit trail consists of a chain of version authenticators and can be used to verify the manner in which the file changed over time. We label the published authenticator P_{v_i}, corresponding to A_{v_i} computed at the file system.

The audit trail may be used to verify the contents of a single version. To audit a single version, the auditor requests version data v_i and the previous version authenticator $A_{v_{i-1}}$ from the file system, computes A_{v_i} using Equation 1 and compares this to the published value P_{v_i}. The computed and published identifiers match if and only if the data currently stored by the file system are identical to the data used to compute the published value. This process verifies the version data content v_i even though $A_{v_{i-1}}$ has not been verified by this audit.

We do not require all version authenticators to be published. A version history (sequence of changes) to a file may be audited based on two published version authenticators separated in time. An auditor accesses two version authenticators P_{v_i} and $P_{v_j}, i < j$. The auditor verifies the individual version v_i with the file system. It then enumerates all versions $v_{i+1}, \ldots, v_j$, computing each version identifier in turn until it computes A_{v_j}. Again, A_{v_j} matches P_{v_j} if and only if the data stored on the file system are identical to the data used to generate the version identifiers, *including all intermediate versions.*

Verifying individual versions and version histories relies upon the collision resistant properties of MACs. For individual versions, the auditor uses the unverified and untrusted $A_{v_{i-1}}$ from the file system. A_{v_i} authenticates version v_i even when an adversary can choose input $A_{v_{i-1}}$. Finding a replacement for $A_{v_{i-1}}$ and v_i that produces the correct A_{v_i}, finds a hash collision. A similar argument allows a version history to be verified based on the authenticators of its first and last version. Finding an alternate version history that matches both endpoints finds a collision.

Version authenticators may be published infrequently. The file system may perform many updates without publication as long as it maintains a local copy of a version authenticator. This creates a natural trade-off between the amount of space and network bandwidth used by the publishing process and the efficiency of verifying version histories. We quantify this trade-off in Section 6.3.

4.1 Incrementally Calculable MACs

I/O efficiency is the principal concern in the calculation and verification of version authenticators in a file system. A version of a file shares data with its predecessor. It differs only in the blocks of data that are changed. As a consequence, the file system performs I/O only on these changed blocks. For performance reasons, it is imperative that the system updates audit trails based only on the changed data. To achieve this, we rely on incremental MAC constructions that allow the MAC of a new version to be calculated using only the previous MAC and the data that have changed. Thus, MAC computation performance scales with the amount of data that are written, rather than size of the file being MACed.

Typical MAC constructions, such as HMAC [3] and CBC-MAC [2, 16], are serial in nature; they require the entire input data to compute the MAC. HMAC relies on a standard hash function H, such as SHA1 [29], which is called twice as

$$H_K(M) = H(K \oplus \text{pad1} || H(K \oplus \text{pad2} || M)).$$

HMAC is very efficient. It costs little more than a single call of the underlying hash function – the outer hash is computed on a very short input. However, HMAC is serial because all data are used as input to the inner hash function. CBC-MAC builds on a symmetric cipher used in CBC mode. In particular, given a message M, divided in blocks $M_1, \ldots M_k$, and a cipher $E_K(\cdot)$, it computes $O_1 = E_K(M_1)$ and $O_i = E_K(M_i \oplus O_{i-1})$, for $2 \leq i \leq k$. CBC-MAC(M) is then the final value O_k. CBC-MAC is inherently serial because the computation of O_i depends on the previous value O_{i-1}.

We use the XOR MAC construction [5], which improves on CBC-MAC, making it incremental and parallelizable. XOR MAC (XMACR in Bellare [5]) builds upon a block cipher $E_K(\cdot)$ in which the block size is n. A message M is divided into blocks, each of a certain length m, as $M = M_1 \ldots M_k$ (M_k is padded if its length is less than m). Then XOR MAC(M) is computed as (r, Z), for a random seed r, and

$$Z = E_K(0||r) \oplus \left[\bigoplus_{j=1}^{k} E_K(1||\langle j \rangle||M_j) \right] \quad (2)$$

in which 0, 1 are bits and $\langle j \rangle$ is the binary representation of block index j. The leading bit differentiates the contribution of the random seed from all block inputs. The inclusion of the block index prevents reordering attacks. Reordering the message blocks results in different authenticators. When using AES-128 [10] for $E_K(\cdot)$, $n = 128$ and $|r| = 127$ bits. When using 47 bits for the block index $\langle j \rangle$, XOR MAC makes an AES call for every 80 bits of the message M.

Our implementation of XOR MAC aligns the block sizes used in the algorithm to that of file system blocks: $|M_j| = 4096$ bytes. As suggested by the original publication [5], a keyed hash function can be used in place of a block cipher to improve performance. We use HMAC-SHA1 to instantiate E_K.

XOR MAC provides several advantages when compared with CBC-MAC or HMAC. It is parallelizable in that the calls to the block cipher can be made in parallel. This is important in high-speed networks, multiprocessor machines, or when out-of-order verification is needed [5], for instance when packets arrive out-of-order owing to loss and retransmission. Most important to our usage, XOR MAC is incremental with respect to block replacement. When message block M_j has been modified into M'_j, it is possible to compute a new MAC (r', Z'), for a fresh random value r', on the entire M by starting from the old value (r, Z) as

$$T = Z \oplus E_K(0||r) \oplus E_K(1||\langle j\rangle||M_j)$$
$$Z' = T \oplus E_K(0||r') \oplus E_K(1||\langle j\rangle||M'_j)$$

XORing out the contributions of the old block and old random seed to make T and XORing in the contributions of the new block and new random seed to build Z'. File systems perform only block replacements. They do not insert or delete data, which would change the alignment of the blocks within a file.

PMAC [6] improves upon XOR MAC in that it makes fewer calls to the underlying block cipher. XOR MAC expands data by concatenating an index to the message block. PMAC avoids this expansion by defining a sequence of distinct *offsets* that are XORed with each message block. Thus, it operates on less data, resulting in fewer calls to the underlying block cipher. Indeed, we initially proposed to use PMAC in our system [7].

However, when XOR MAC or PMAC are instantiated with keyed hash functions (rather than block ciphers), the performance benefits of PMAC are minimal for file systems. The reason is that HMAC-SHA1 accepts large inputs, permitting the use of a 4096 byte file system block. The incremental cost of a 64 bit expansion, representing a block index, is irrelevant when amortized over a 4096 byte block. At the same time, XOR MAC is simpler than PMAC and easier to implement. (On the other hand, PMAC is deterministic, requires no random inputs, and produces smaller output). In our system, we elect to implement XOR MAC.

4.2 XOR MAC for Audit Trails

We use the incremental property of XOR MAC to perform block-incremental computation for copy-on-write file versions. Each version v_i comprises blocks $b_{v_i}(1), \ldots, b_{v_i}(n)$ equal to the file system block size and a file system independent representation of the version's metadata, denoted $\overline{M}_{v_i}$ (see Section 4.3). The output of XOR MAC is the exclusive-or of the one-way functions

$$\begin{aligned} A_{v_i} =& H_K(00||r_{v_i}) \oplus \left[\bigoplus_{j=1}^{n} H_K(01||\langle j\rangle||b_{v_i}(j))\right] \\ &\oplus H_K(10||\overline{M}_{v_i}) \oplus H_K(11||A_{v_{i-1}}) \end{aligned} \tag{3}$$

in which r_{v_i} is a random number unique to version v_i. This adapts equation 2 to our file system data. We have added an additional leading bit that allows for four distinct components to the input. Bit sequences 00, 01, and 10 precede the random seed, block inputs, and normalized metadata respectively. To these, we add the previous version authenticator, which forms the version hash chain defined by equation 1. This form is the full computation and is stored as the pair (r, A_{v_i}). There is also an incremental computation. Assuming that version v_i differs from v_{i-1} in one block only $b_{v_i}(j) \neq b_{v_{i-1}}(j)$, we observe that

$$\begin{aligned} A_{v_i} =& A_{v_{i-1}} \oplus H_K(00||r_{v_i}) \oplus H_K(00||r_{v_{i-1}}) \\ &\oplus H_K(01||\langle j\rangle||b_{v_i}(j)) \oplus H_K(01||\langle j\rangle||b_{v_{i-1}}(j)) \\ &\oplus H_K(10||\overline{M}_{v_i}) \oplus H_K(10||\overline{M}_{v_{i-1}}) \\ &\oplus H_K(11||A_{v_{i-1}}) \oplus H_K(11||A_{v_{i-2}}). \end{aligned}$$

This extends trivially to any number of changed blocks. The updated version authenticator adds the contribution of the changed blocks and removes the contribution of those blocks in the previous version. It also updates the contributions of the past version authenticator, normalized metadata, and random seed.

The computation of XOR MAC authenticators scales with the amount of I/O, whereas the performance of a hash message authentication code (HMAC) scales with the file size. With XOR MAC, only new data being written to a version will be authenticated. HMACs must process the entire file, irrespective of the amount of I/O. This is problematic as studies of versioning file systems show that data change at a fine granularity [31, 38]. Our results (Section 6) confirm the same. More importantly, the computation of the XOR MAC version authenticator requires only those data blocks being modified, which are already in cache, requiring little to no additional disk I/O. Computing an HMAC may require additional I/O. This is because system caches are managed on a page basis, leaving unmodified and unread portions of an individual file version on disk. When computing an HMAC for a file, all file data would need to be accessed. As disk accesses are a factor of 10^5 slower than memory accesses, computing an HMAC may be substantially worse than algorithmic performance would indicate.

The benefits of incremental computation of MACs apply to both writing data and conducting audits. When versions of a file share much data in common, the differences between versions are small, allowing for efficient version verification. Incremental MACs allow an auditor to authenticate the next version by computing the authenticity of only the data blocks that have changed. When performing an audit, the authenticity of the entire version history may be determined by a series of small, incremental computations. HMACs do not share this advantage and must authenticate all data in all versions.

4.3 File System Independence

Many storage management tasks alter a file system, including the metadata of past versions, but should not result in an audit failure. Examples include: file-oriented restore of backed-up data after a disk failure, resizing or changing the logical volumes underlying a file system, compaction/defragmentation of storage, and migration of data from one file system to another. Thus, audit models must be robust to such changes. We call this property *file system independence*. Audit information is bound to the file data and metadata and remains valid when the physical implementation of a file changes. This includes transfers of a file from system to system (with the caveat that all systems storing data support audit trails – we have implemented only one). The act of performing a data restoration may be a procedure worth auditing in and of itself. We consider this outside the scope of the file system requirements.

Our authenticators use the concept of *normalized metadata* for file system independence. Normalized metadata are the persistent information that describe attributes of a file system object independent of the file system architecture. These metadata include: name, file size, ownership and permissions, and modification, creation and access times. These fields are common to most file systems and are stored persistently with every file. Normalized metadata do not include physical offsets and file system specific information, such as inode number, disk block addresses, or file system flags. These fields are volatile in that storage management tasks change their values. Normalized metadata are included in authenticators and become part of a file's data for the purposes of audit trails.

4.4 Hierarchies and File Systems

Audit trails must include information about the entire state of the file system at a given point in time. Auditors need to discover the relationships between files and interrogate the contents of the file system. Having found a file of interest in an audit, natural questions include: what other data was in the same directory at this time? or, did other files in the system store information on the same topic? The data from each version must be associated with a coherent view of the entire file system.

Authenticating directory versions as if they were file versions is insufficient. A directory is a type of file in which the data are directory entries (name-inode number pairs) used for indexing and naming files. Were we to use our previous authenticator construction (Equation 3), a directory authenticator would be the MAC of its data (directory entries), the MAC of the previous directory authenticator and its normalized inode information. However, this construct fails to bind the data of a directory's files to the names, allowing an attacker to undetectably exchange the names of files within a directory.

We employ trees of MACs that bind individual versions and their names to a file system hierarchy, authenticating the entire versioning file system recursively. In addition to the normalized inode information and previous authenticator used to authenticate files, directory authenticators are composed of name-authenticator pairs. For each file within the directory, we concatenate its authenticator to the corresponding name and take a one-way hash of the result.

$$
\begin{aligned}
A_{D_i} = & H_K(00||r_{D_i}) \oplus \left[\bigoplus_{j=1}^{n} H_K(01||\langle j\rangle||name_j||A_{v_j}) \right] \\
& \oplus H_K(10||\overline{M}_{D_i}) \oplus H_K(11||A_{D_{i-1}})
\end{aligned}
$$

This binds files and sub-directories to the authenticator of the parent directory. Directory version authenticators continue recursively to the file system root, protecting the entire file system image. The SFS-RO system [13] used a similar technique to fix the contents of a read-only file system without versioning. Our method differs in that it is incremental and accounts for updates.

For efficiency reasons, we bind versions to the directory's authenticator lazily. Figure 1 shows how directory D binds to files S, T, U. This is done by including the authenticators for specific versions S_1, T_2, U_4 that were current at the time version D_2 was created. However, subsequent file versions (*e.g.* S_2, T_3) may be created without updating the directory version authenticator A_{D_2}. The system updates the directory authenticator only when the directory contents change; *i.e.*, files are created, destroyed, or renamed. In this example, when deleting file U (Figure 1), the authenticator is updated to the current versions. Alternatively, were we to bind directory version authenticators directly to the content of the most recent file version, they would need to be updated every time that a file is written. This includes all parent directories recursively to the file system root – an obvious performance concern as it would need to be done on every write.

Binding a directory authenticator to a file version binds it to all subsequent versions of that file, by hash chaining of the file versions. This is limited to the portion of the file's version chain within the scope of the directory. A rename moves a file from one directory's scope to another. Ext3cow employs timestamps for version numbers, which can be used to identify the valid file versions within each directory version.

Updating directory authenticators creates a time-space trade-off similar to that of publication frequency (see Section 4). When auditing a directory at a given point

$$\begin{aligned}A_{D_2} =& H_K(00||r_{D_2}) \oplus H_K(01||\langle 1\rangle||name(S)||A_{S_1})\\ &\oplus H_K(01||\langle 2\rangle||name(T)||A_{T_2})\\ &\oplus H_K(01||\langle 3\rangle||name(U)||A_{U_4})\\ &\oplus H_K(10||\overline{M}_{D_2}) \oplus H_K(11||A_{D_1})\end{aligned}$$

$$\begin{aligned}A_{D_3} =& H_K(00||r_{D_3}) \oplus H_K(01||\langle 1\rangle||name(S)||A_{S_4})\\ &\oplus H_K(01||\langle 2\rangle||name(T)||A_{T_5})\\ &\oplus H_K(10||\overline{M}_{D_3}) \oplus H_K(11||A_{D_2})\end{aligned}$$

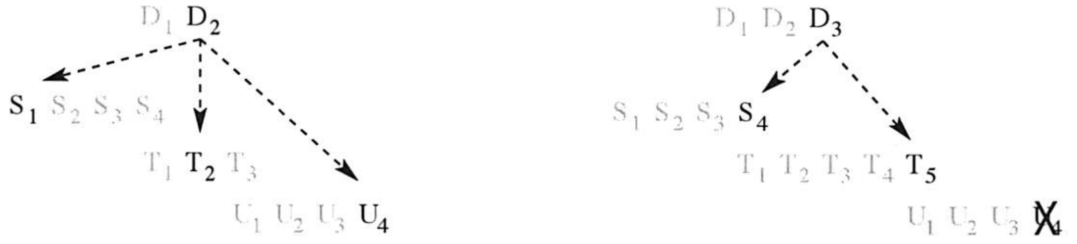


Figure 1: Directory version authenticators before and after file U is deleted. Equations show the full (not incremental) computation.

in time, the auditor must access the directory at the time when its was created and then follow the children files' hash chains forward to the specified point in time. Updating directory authenticators more frequently may be desirable to speed the audit process.

5 File System Implementation

We have implemented digital audit trails using XOR MAC in ext3cow [31], an open-source, block-versioning file system designed to meet the requirements of electronic records legislation. Ext3cow supports file system snapshot, per-file versioning, and a time-oriented interface. Versions of a file are implemented by chaining inodes together in which each inode represents a version. The file system traverses the inode chain to generate a point-in-time view of a file. Ext3cow provides the features needed for an implementation of audit trails: it supports continuous versioning, creating a new version on every write, and maintains old and new versions of data and metadata concurrently for the incremental computation of version authenticators. We store version authenticators for a file in its inode. We have already retrofitted the metadata structures of ext3cow to support versioning and secure deletion (based on authenticated encryption [32]). Version authenticators are a straightforward extension to ext3cow's already augmented metadata, requiring only a few bytes per inode.

5.1 Metadata for Authentication

Metadata in ext3cow have been improved to support incremental versioning authenticators for electronic audit trails. To accomplish this, ext3cow "steals" a single data block pointer from the inode, replacing it with an authentication block pointer, *i.e.* a pointer to disk block holding authentication information. Figure 2 illustrates the metadata architecture. The number of direct blocks has been reduced by one, from twelve to eleven, for storing an authenticator block (`i_data[11]`). Block stealing for authenticators reduces the effective file size by only one file system block, typically 4K.

Each authenticator block stores five fields: the current version authenticator (A_{v_i}), the authenticator for the previous version ($A_{v_{i-1}}$), the one-way hash of the authenticator for the previous version ($H_K(11||A_{v_{i-1}})$), the authenticator for the penult-previous version ($A_{v_{i-2}}$), and the the one-way hash of the authenticator for the penult-previous version ($H_K(11||A_{v_{i-2}})$). Each authenticator computation requires access to the previous and penult-previous authenticators and their hashes. By storing authenticators and hashes for previous versions together, the system avoids two read I/Os: one for each previous version authenticator and hash computations. When a new version is generated and a new inode is created, the authenticator block is copy-on-written and "bumps" each entry; *i.e.*, copying the once current authenticator (A_{v_i}) to the previous authenticator ($A_{v_{i-1}}$), and the previous authenticator ($A_{v_{i-1}}$) and hash ($H_K(11||A_{v_{i-1}})$) to the penult-previous authenticator ($A_{v_{i-2}}$) and hash ($H_K(11||A_{v_{i-2}})$). The once current authenticator (A_{v_i}) is zeroed, and is calculated on an as-needed basis.

In almost all cases, authenticator blocks do not increase the number of disk seeks performed by the system. The block allocator in ext3cow makes efforts to collocate data, metadata, and authenticator blocks in a single disk drive track, maintaining contiguity. Authenticator blocks are very likely to be read out of the disk's track cache. The same disk movement that reads inode or data blocks populates the track cache.

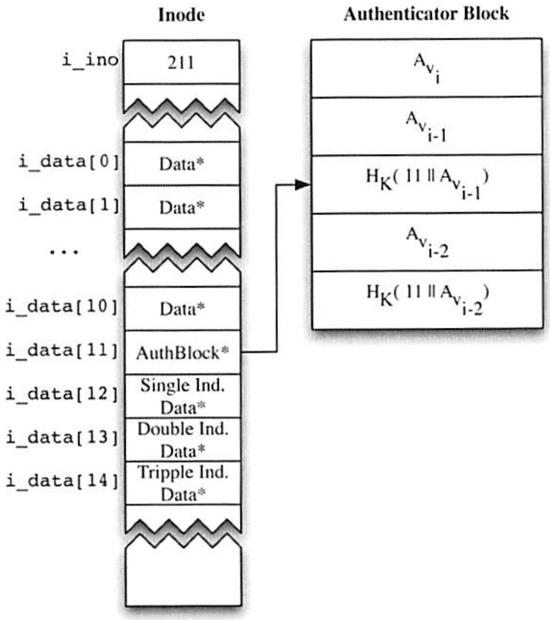


Figure 2: Metadata architecture to support version authenticators.

5.2 Key Management

Key management in ext3cow uses lockboxes [17] to store a per-file authentication key. The file owner's private key unlocks the lockbox and provides access to the authentication key. Lockboxes were developed as part of the authenticated encryption and secure deletion features of ext3cow [32].

6 Experimental Results

We measure the impact of authentication on versioning file systems and compare the performance characteristics of HMAC and XOR MAC in the ext3cow versioning file system. We begin by comparing the CPU and disk throughput performance of HMAC and XOR MAC by using two micro-benchmarks: one designed to contrast the maximum throughput capabilities of each algorithm and one designed to highlight the benefits of the incremental properties of XOR MAC. We then use a traced file system workload to illustrate the aggregate performance benefits of incremental authentication in a versioning file system. Lastly, we use file system traces to characterize some of the overheads of generating authenticators for the auditing environment. Both authentication functions, XOR MAC and HMAC, were implemented in the ext3cow file system using the standard HMAC-SHA1 keyed-hash function provided by the Linux kernel cryptographic API [25]. For brevity, XOR MAC implemented with HMAC-SHA1 is further referred to as XOR MAC-SHA1. All experiments were performed on a Pentium 4, 2.8GHz machine with 1 gigabyte of RAM. Trace experiments were run on a 80 gigabyte ext3cow partition of a Seagate Barracuda ST380011A disk drive.

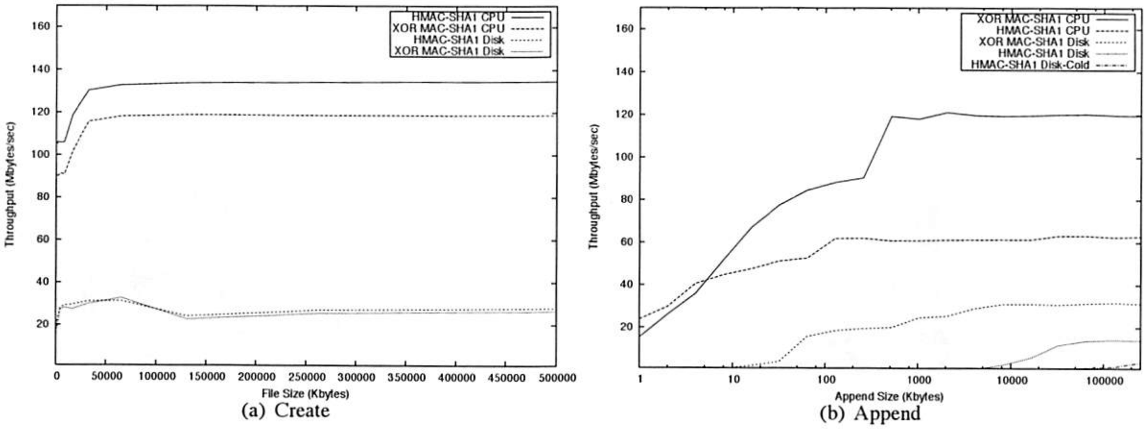
6.1 Micro-benchmarks

To quantify the efficiency of XOR MAC, we conducted two micro-benchmark experiments: *create* and *append*. The *create* test measures the throughput of creating and authenticating files of size 2^N bytes, where $N = 0, 1, \ldots, 30$ (1 byte to 1 gigabyte files). The test measures both CPU throughput, *i.e.* the time to calculate a MAC, and disk throughput, *i.e.* the time to calculate a MAC and write the file to disk. Files are created and written in their entirety. Thus, there are no benefits from incremental authentication. The *append* experiment measures the CPU and disk throughput of appending 2^N bytes to the same file and calculating a MAC, where $N = 0, 1, \ldots, 29$ (1 byte to 500 megabytes). For XOR MAC, an append requires only a MAC of a new random value, a MAC of each new data block and an XOR of the results with the file's authenticator. HMAC does not have this incremental property and must MAC the entire file data in order to generate the correct authenticator, requiring additional read I/O. We measure both warm and cold cache configurations. In a warm cache, previous appends are still in memory and the read occurs at memory speed. In practice, a system does not always find all data in cache. Therefore, the experiment was also run with a cold cache; before each append measurement, the cache was flushed.

Figure 3(a) presents the results of the *create* micro-benchmark. Traditional HMAC-SHA1 has higher CPU throughput than XOR MAC-SHA1, saturating the CPU at 134.8 MB/s. The XOR MAC achieves 118.7 MB/s at saturation. This is expected as XOR MAC-SHA1 performs two calls to SHA1 for each block (see Equation 3), compared to HMAC-SHA1 that only calls SHA1 twice for each file, resulting in additional computation time. Additionally, SHA1 appends the length of the message that it's hashing to the end of the message, padding up to 512-bit boundaries. Therefore, XOR MAC-SHA1 hashes more data, up to n*512 bits more for n blocks.

Despite XOR MAC's computational handicap, disk throughput measurements show little performance disparity. HMAC-SHA1 achieves a maximum of 28.1 MB/s and XOR MAC-SHA1 a maximum of 26.6 MB/s. This illustrates that calculating new authenticators for a file system is I/O-bound, making XOR MAC-SHA1's ultimate performance comparable to that of HMAC-SHA1.

The results of the *append* micro-benchmark make a compelling performance argument for incremental MAC



(a) Create

(b) Append

Figure 3: Results of micro-benchmarks measuring the CPU and disk throughput.

computation. Figure 3(b) shows these results – note the log scale. We observe XOR MAC-SHA1 outperforms HMAC-SHA1 in both CPU and disk throughput measurements. XOR MAC-SHA1 bests HMAC-SHA1 CPU throughput, saturating at 120.3 MB/s, compared to HMAC-SHA1 at 62.8 MB/s. Looking at disk throughput, XOR MAC-SHA1 also outperforms the best-case of an HMAC calculation, warm-cache HMAC-SHA1, achieving a maximum 31.7 MB/s, compared to warm-cache HMAC-SHA1 at 20.9 MB/s and cold-cache HMAC-SHA1 at 9.7 MB/s. These performance gains arise from the incremental nature of XOR MACs. In addition to the extra computation to generate the MAC, an ancillary read I/O is required to bring the old data into the MAC buffer. While the *append* benchmark is contrived, it is a common I/O pattern. Many versioning file systems implement versioning with a copy-on-write policy. Therefore, all I/O that is not a full overwrite is, by definition, incremental and benefits from the incremental qualities of XOR MAC.

6.2 Aggregate Performance

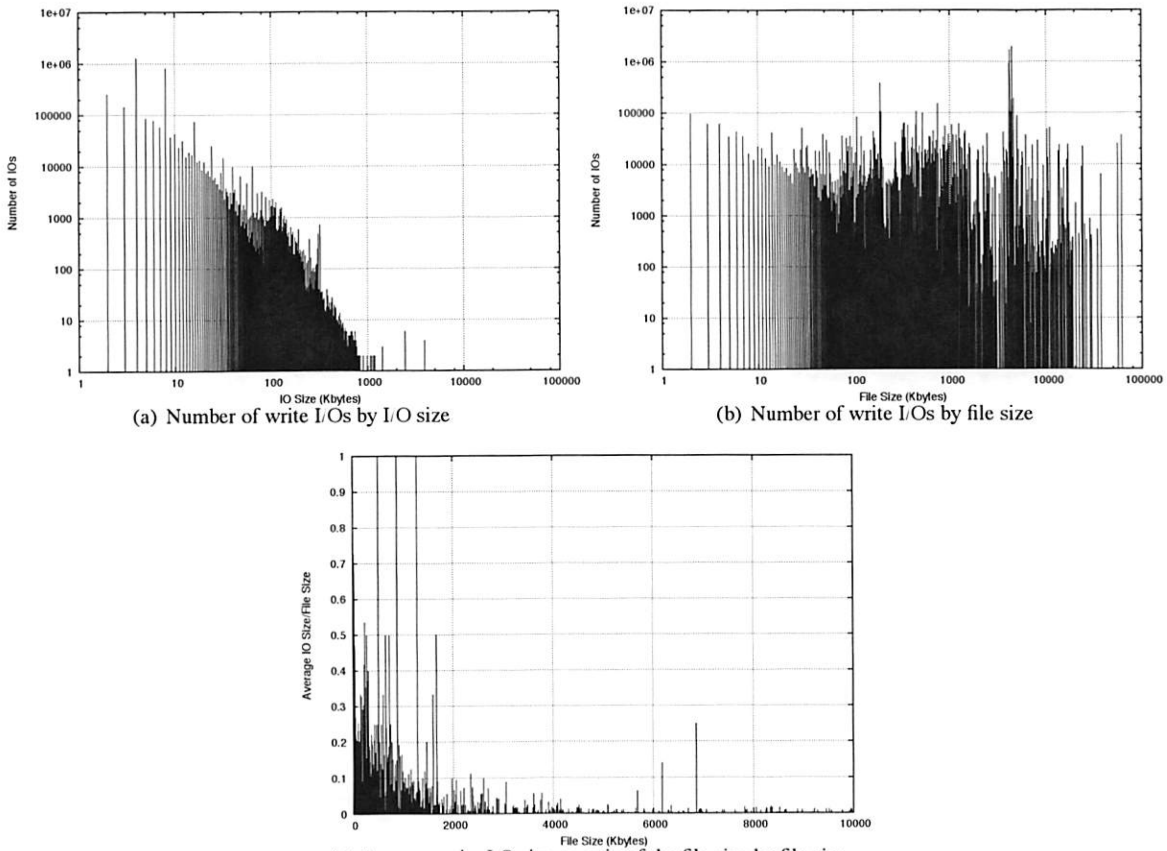
We take a broader view of performance by quantifying the aggregate benefits of XOR MAC on a versioning file system. To accomplish this, we replayed four months of system call traces [35] on an 80 gigabyte ext3cow partition, resulting in 4.2 gigabytes of data in 81,674 files. Despite their age, these 1996 traces are the most suitable for these measurements. They include information that allow multiple open/close sessions on the same file to be correlated – necessary information to identify versioning. More recent traces [11, 19, 36, 38] do not include adequate information to correlate open/close sessions, are taken at too low a level in the IO system to be useful, or would introduce new ambiguities, such as the effects of a network file system, into the aggregate measurements. Our experiments compare trace-driven throughput performance as well as the total computation costs for performing a digital audit using the XOR MAC and HMAC algorithms. We analyze aggregate results of run-time and audit performance and examine how the incremental computation of MACs benefits copy-on-write versioning.

No Authentication	XOR MAC	HMAC
1.98 MB/s	1.77 MB/s	0.11 MB/s

Table 1: The trace-driven throughput of no authentication, XOR MAC and HMAC.

6.2.1 Write Performance

The incremental computation of XOR MAC minimally degrades on-line system performance when compared with a system that does not generate audit trails (No Authentication). In contrast, HMAC audit trails reduce throughput by more than an order of magnitude (Table 1). We measure the average throughput of the system while replaying four months of system call traces. The traces were played as fast as possible in an effort to saturate the I/O system. The experiment was performed on ext3cow using no authentication, HMAC-SHA1 authentication, and XOR MAC-SHA1 authentication. XOR MAC-SHA1 achieves a 93.9% improvement in run-time performance over HMAC-SHA1: 1.77 MB/s versus 0.11 MB/s. HMAC-SHA1's degradation results from the additional read I/O and computation time it must perform on every write. XOR MAC-SHA1 incurs minimal performance penalties owing to its ability to compute authenticators using in-cache data. XOR MAC-SHA1 achieves 89% of the throughput of a system with no authentication.



(a) Number of write I/Os by I/O size

(b) Number of write I/Os by file size

(c) Average write I/O size as ratio of the file size by file size

Figure 4: Characterization of write I/Os from trace-driven experiments.

To better understand the run-time performance differences between XOR MAC and HMAC, we characterize the number and size of writes and how they are written to various files in the system. By looking at each write request as a function of its destination file size, we can see why incremental computation of MACs is beneficial to a file system. Our observations confirm three things: (1) Most write requests are small, (2) write requests are evenly distributed among all file sizes, and (3) the size of write requests are usually a tiny fraction of the file size. Figure 4(a) presents statistics on the number and size of write I/Os, whereas Figure 4(b) shows number of write I/Os performed by file size. Both plots are log-log. We observe that of the 16,601,128 write I/Os traced over four months, 99.8% of the I/Os are less than 100K, 96.8% are less than 10K, and 72.4% are less than 1K in size. This shows that a substantial number of I/Os are small. We also observe that files of all sizes receive many writes. Files as large as 100 megabytes receive as many as 37,000 writes over the course of four months. Some files, around 5MB in size, receive nearly two million I/Os. These graphs show that I/O sizes are, in general, small and that files of all sizes receive many I/Os.

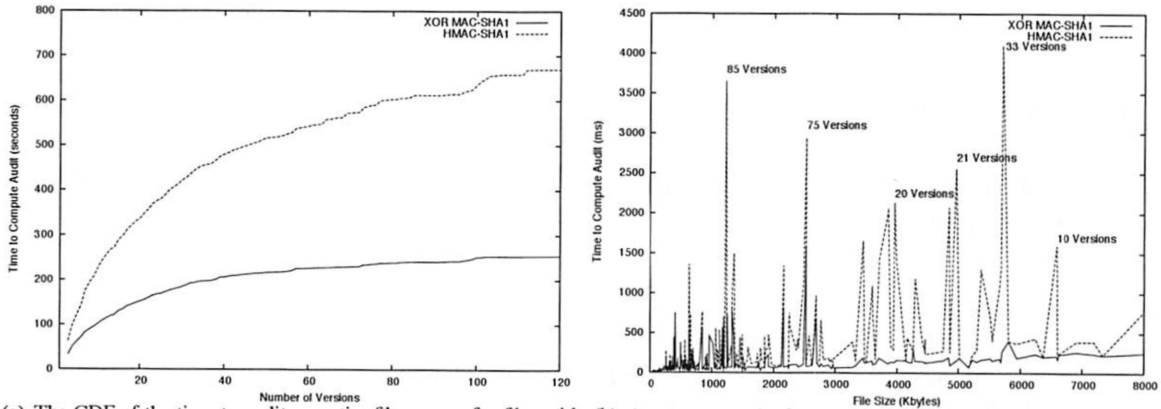
The relationship between I/O size and file size reveals the necessity of incremental MAC computation. Figure 4(c) presents the average write I/O size as a ratio of the file size over file sizes. This plot shows that there are few files that receive large writes or entire overwrites in a single I/O. In particular, files larger than 2MB receive writes that are a very small percentage of their file size. The largest files receive as little as 0.025% of their file size in writes and nearly all files receive less than 25% of their file size in write I/Os. It is this disproportionate I/O pattern that benefits the incremental properties of XOR MAC. When most I/Os received by large files are small, a traditional HMAC suffers in face of additional computation time and supplementary I/Os. The performance of XOR MAC, however, is immune to file size and is a function of write size alone.

6.2.2 Audit Performance

To generate aggregate statistics for auditing, we aged the file system by replaying four months of traced system calls, taking snapshots daily. We then performed two audits of the file system, one using HMAC-SHA1 and one

Number of Versions	HMAC-SHA1 (seconds)	XOR MAC-SHA1 (seconds)
All	11209.4	10593.1
≥ 2	670.1	254.4

Table 2: The number of seconds required to audit an entire file system using HMAC-SHA1 and XOR MAC-SHA1 for all files and only those files with two or more versions.



(a) The CDF of the time to audit an entire file system for files with two or more versions by number of versions

(b) Aggregate results for auditing an entire file system by file size

Figure 5: Aggregate auditing performance results for XOR MAC-SHA1 and HMAC-SHA1.

using XOR MAC-SHA1. Our audit calculated authenticators for every version of every file. Table 2 presents the aggregate results for performing an audit using XOR MAC-SHA1 and HMAC-SHA1. The table shows the result for all files and the result for those files with two or more versions. Auditing the entire 4.2 gigabytes of file system data using standard HMAC-SHA1 techniques took 11,209 seconds, or 3.11 hours. Using XOR MAC-SHA1, the audit took 10,593 seconds, or 2.94 hours; a savings of 5% (10 minutes).

Most files in the trace (88%) contain a single version, typical of user file systems. These files dominate audit performance and account for the similarity of HMAC and XOR MAC results. However, we are interested in file systems that contain medical, financial, and government records and, thus, will be populated with versioned data. To look at auditing performance in the presence of versions, we filter out files with only one version. On files with two or more versions, XOR MAC-SHA1 achieves a 62% performance benefit over HMAC-SHA1, 670 versus 254 seconds. A CDF of the time to audit files by number of versions is presented in Figure 5(a). XOR MAC-SHA1 achieves a 37% to 62% benefit in computation time over HMAC-SHA1 for files with 2 to 112 versions. This demonstrates the power of incremental MACs when verifying long version chains. The longer the version chain and the more data in common, the better XOR MAC performs.

Looking at audit performance by file size shows that the benefit is derived from long version chains. Figure 5(b) presents a break down of the aggregate audit results by file size. There exists no point at which XOR MAC-SHA1 performs worse than HMAC-SHA1, only points where they are the same or better. Performance is the same for files that have a single version and for files that do not share data among versions. As the number of versions increase and much data are shared between versions, large discrepancies in performance arise. Some examples of files with many versions that share data are annotated. XOR MAC shows little performance variance with the number of versions.

6.3 Requirements for Auditing

As part of our audit model, authenticators are transfered to and stored at a third party. We explore the storage and bandwidth resources that are required for version authentication. Four months of file system traces were replayed over different snapshot intervals. At a snapshot, authentication data are transfered to the third party, committing the file system to that version history. Measurements were taken at day, hour, and minute snapshot intervals. During each interval, the number of file modifications and number of authenticators generated were captured.

Figure 6 presents the size of authentication data generated over the simulation time for the three snapshot in-

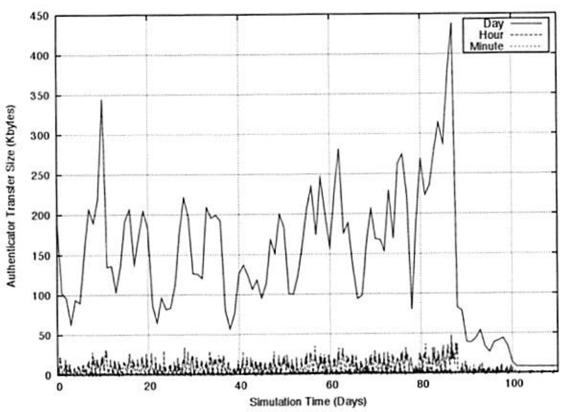


Figure 6: Size of authentication data from four months of traced workloads at three snapshot intervals.

tervals. Naturally, the longer the snapshot interval, the larger the number of authenticators generated. However, authentication data are relatively small; even on a daily snapshot interval, the largest transfer is 450K, representing about 22,000 modified files. Authenticators generated by more frequent snapshot (hourly or per-minute) never exceed 50KB per transfer. Over the course of four months, a total of 15.7MB of authentication data are generated on a daily basis from 801,473 modified files, 22.7MB on a hourly basis from 1,161,105 modified files, and 45.4MB on a per-minute basis from 2,324,285 modified files. The size of authenticator transfer is invariant of individual file size or total file system size; it is directly proportional to the number of file modifications made in a snapshot interval. Therefore, the curves in Figure 6 are identical to a figure graphing the number of files modified over the same snapshot intervals.

7 Future Work

Conducting digital audits with version authenticators leaves work to be explored. We are investigating authentication and auditing models that do not rely on trusted third parties. We also discuss an entirely different model for authentication based on approximate MACs, which can tolerate partial data loss.

7.1 Alternative Authentication Models

Having a third party time-stamp and store a file system's authenticators may place undue burden, in terms of storage capacity and management, on the third party. Fortunately, it is only one possible model for a digital auditing system. We are currently exploring two other possible architectures for managing authentication data; a storage-less third party and cooperative authentication. In a storage-less third party model a file system would generate authenticators and transmit them to a third party. Instead of storing them, the third party would MAC the authenticators and return them to the file system. The file system stores both the original authenticators and those authenticated by the third party. In this way, the third party stores nothing but signing keys, placing the burden of storing authenticators on the file system. When the file system is audited, the auditor requests the signing keys from the third party and performs two authentication steps: first, checking the legitimacy of the stored authenticators and then checking the authenticity of the data themselves.

This design has limitations. The scheme doubles the amount of authentication data transfered. Additionally, because the third party keeps no record of any file, an attacker may delete an entire file system without detection or maintain multiple file systems, choosing which file system to present at audit time. Portions of the file system may not be deleted or modified, because the authenticators for version chains and directory hierarchies bind all data to the file system root authenticator.

A further variant groups peers of file systems together into a cooperative ring, each storing their authentication data on an adjoining file system. A file system would store the previous system's authenticator in a log file, which is subsequently treated as data, resulting in the authenticators being authenticated themselves. This authenticator for the log file is stored on an adjoining system, creating a ring of authentication. This design relieves the burden on a single third party from managing all authentication data and removes the single point of failure for the system. This architecture also increases the complexity of tampering by a factor of N, the number of links of in the chain. Because an adjoining file system's authenticators are kept in a single log file, only one authenticator is generated for that entire file system, preventing a glut of authentication data.

7.2 Availability and Security

A verifiable file system may benefit from accessing only a portion of the data to establish authenticity. Storage may be distributed across unreliable sites [9, 18], such that accessing it in it's entirety is difficult or impossible. Also, if data from any portion of the file system are corrupted irreparably, the file system may still be authenticated, whereas with standard authentication, altering a single bit of the input data leads to a verification failure.

To audit incomplete data, we propose the use approximately-secure and approximately-correct MAC (AMAC) introduced by Di Crescenzo *et al.* [8]. The system verifies authenticity while tolerating a small amount of modification, loss, or corruption of the original data.

We propose to make the AMAC construction incremental to adapt it to file systems; in addition, we plan to use XOR MAC as a building block in the AMAC construction [8], to allow for incremental update. The atom for the computation is a file system block, rather than a bit. The approximate security and correctness then refer to the number of corrupted or missing blocks, rather than bits. The exact level of tolerance may be tuned.

The chief benefit of using the AMAC construction over regular MAC constructions lies in verification. Serial and parallel MACs require the entire message as input to verify authenticity. Using AMAC, a portion of the original message can be ignored. This allows a weaker statement of authenticity to be constructed even when some data are unavailable. The drawback of AMAC lies in the reduction of authenticity. With AMAC, some data may be acceptably modified in the original source.

8 Conclusions

We have introduced a model for digital audits of versioning file systems that supports compliance with federally mandated data retention guidelines. In this model, a file system commits to a version history by transmitting audit metadata to a third party. This prevents the owner of the file system (or a malicious party) from modifying past data without detection. Our techniques for the generation of audit metadata use incremental authentication methods that are efficient when data modifications are fine grained, as in versioning file systems. Experimental results show that incremental authentication can perform up to 94% faster than traditional serial authentication algorithms. We have implemented incremental authentication in ext3cow, an open-source versioning file system, available at: *www.ext3cow.com.*

9 Acknowledgments

This work was supported by the National Science Foundation (awards CCF-0238305 and IIS-0456027), by the Department of Energy, Office of Science (award DE-FG02-02ER25524), and by the IBM Corporation. We thank Giovanni Di Crescenzo for discussions on the AMAC construction.

References

[1] ADAMS, C., CAIN, P., PINKAS, D., AND ZUCCHERATO, R. IETF RFC 3161 Time-Stamp Protocol (TSP). IETF Network Working Group, 2001.

[2] AMERICAN BANKERS ASSOCIATION. American national standard for financial institution message authentication (wholesale). ANSI X9.9, 1986.

[3] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *Advances in Cryptology - Crypto'96 Proceedings* (1996), vol. 1109, Springer-Verlag, pp. 1–19. Lecture Notes in Computer Science.

[4] BELLARE, M., GOLDREICH, O., AND GOLDWASSER, S. Incremental cryptography and application to virus protection. In *Proceedings of the ACM Symposium on the Theory of Computing* (May-June 1995), pp. 45–56.

[5] BELLARE, M., GUÉRIN, R., AND ROGAWAY, P. XOR MACs: New methods for message authentication using finite pseudorandom functions. In *Advances in Cryptology - Crypto'95 Proceedings* (1995), vol. 963, Springer-Verlag, pp. 15–28. Lecture Notes in Computer Science.

[6] BLACK, J., AND ROGAWAY, P. A block-cipher mode of operation for parallelizable message authentication. In *Advances in Cryptology - Eurocrypt'02 Proceedings* (2002), vol. 2332, Springer-Verlag, pp. 384 – 397. Lecture Notes in Computer Science.

[7] BURNS, R., PETERSON, Z., ATENIESE, G., AND BONO, S. Verifiable audit trails for a versioning file system. In *Proceedings of the ACM CCS Workshop on Storage Security and Survivability* (November 2005), pp. 44–50.

[8] CRESCENZO, G. D., GRAVEMAN, R., GE, R., AND ARCE, G. Approximate message authentication and biometric entity authentication. In *Proceedings of Financial Cryptography and Data Security* (February-March 2005).

[9] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (October 2001), pp. 202–215.

[10] DAEMEN, J., AND RIJMEN, V. *The Design of Rijndael: AES – the Advanced Encryption Standard.* Springer, 2002.

[11] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive NFS tracing of email and research workloads. In *Proceedings of the USENIX File and Storage Technologies Conference (FAST)* (March 2003), pp. 203–216.

[12] FARMER, D., AND VENEMA, W. *Forensic Disocvery.* Addison-Wesley, 2004.

[13] FU, K., KASSHOEK, M. F., AND MAZIÈRES, D. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems 20*, 1 (2002), 1–24.

[14] GIFFORD, D. K., NEEDHAM, R. M., AND SCHROEDER, M. D. The Cedar file system. *Communications of the ACM 31*, 3 (March 1988), 288–298.

[15] HAUBERT, E., TUCEK, J., BRUMBAUGH, L., AND YURCIK, W. Tamper-resistant storage techniques for multimedia systems. In *IS&T/SPIE Symposium Electronic Imaging Storage and Retrieval Methods and Applications for Multimedia (EI121)* (January 2005), pp. 30–40.

[16] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. Information technology - security techniques - data integrity mechanism using a cryptographic check function employing a block cipher algorithm. ISO/IEC 9797, April 1994.

[17] Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., and Fu, K. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2003), pp. 29–42.

[18] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummandi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the ACM Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)* (November 2000), pp. 190–201.

[19] Kuenning, G. H., Popek, G. J., and Reihe, P. An analysis of trace data for predictive file caching in mobile computing. In *Proceedings of the Summer USENIX Technical Conference* (June 1994).

[20] Lamport, L. Password authentication with insecure comunication. *Communications of the ACM 24*, 11 (1981), 770–772.

[21] Maniatis, P., and Baker, M. Enabling the archival storage of signed documents. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (January 2002), pp. 31–46.

[22] McCoy, K. *VMS File System Internals.* Digital Press, 1990.

[23] Micali, S. Efficient certificate revocation. Tech. Rep. MIT/LCS/TM-542b, Massachusetts Institute of Technology, 1996.

[24] Monroe, J. Emerging solutions for content storage. Presentation at PlanetStorage, 2004.

[25] Morris, J. The Linux kernel cryptographic API. *Linux Journal*, 108 (April 2003).

[26] Muniswamy-Reddy, K.-K., Holand, D. A., Braun, U., and Seltzer, M. Provenance-aware storage systems. In *Proceedings of the USENIX Annual Technical Conference* (June 2006).

[27] Muniswamy-Reddy, K.-K., Wright, C. P., Himmer, A., and Zadok, E. A versatile and user-oriented versioning file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2004), pp. 115–128.

[28] National Institute of Standards and Technology. Digital signature standard (DSS). Federal Information Processing Standards (FIPS) Publication 186, May 1994.

[29] National Institute of Standards and Technology. Secure hash standard. Federal Information Processing Standards (FIPS) Publication 180-1, April 1995.

[30] Patil, S., Kashyap, A., Sivathanu, G., and Zadok, E. I^3FS: An in-kernel integrity checker and intrusion detection file system. In *Proceedings of the Large Installation System Administration Conference (LISA)* (November 2004), pp. 67–78.

[31] Peterson, Z., and Burns, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transcations on Storage 1*, 2 (2005), 190–212.

[32] Peterson, Z. N. J., Burns, R., Herring, J., Stubblefield, A., and Rubin, A. Secure deletion for a versioning file system. In *Proceedings of the USENIX Conference on File And Storage Technologies (FAST)* (December 2005), pp. 143–154.

[33] Quinlan, S., and Dorward, S. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File And Storage Technologies (FAST)* (January 2002), pp. 89–101.

[34] Rivest, R. L. All-or-nothing encryption and the package transform. In *Proceedings of the Fast Software Encryption Conference* (1997), vol. 1267, pp. 210–218. Lecture Notes in Computer Science.

[35] Roselli, D., and Anderson, T. E. Characteristics of file system workloads. Research report, University of California, Berkeley, June 1996.

[36] Roselli, D., Lorch, J., and Anderson, T. A comparison of file system workloads. In *Proceedings of the USENIX Technical Conference* (2000), pp. 41–54.

[37] Schneier, B., and Kelsey, J. Secure audit logs to support computer forensics. *ACM Transactions on Information Systems Security 2*, 2 (1999), 159–176.

[38] Soules, C. A. N., Goodson, G. R., Strunk, J. D., and Ganger, G. R. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2003), pp. 43–58.

[39] United States Congress. The Health Insurance Portability and Accountability Act (HIPAA), 1996.

[40] United States Congress. The Sarbanes-Oxley Act (SOX). 17 C.F.R. Parts 228, 229 and 249, 2002.

[41] Waldman, M., Rubin, A. D., and Cranor, L. F. Publius: A robust, tamper-evident, censorship-resistant, Web publishing system. In *Proceedings of the USENIX Security Symposium* (August 2000), pp. 59–72.

[42] Weatherspoon, H., Wells, C., and Kubiatowicz, J. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Proceedings of the Workshop on Future Directions in Distributed Computing* (June 2002), pp. 142–147.

Architectures for Controller Based CDP

Guy Laden *Paula Ta-Shma* *Eitan Yaffe* *Michael Factor*
Shachar Fienblit
IBM Haifa Research Laboratories
{laden, paula, eitany, factor, shachar }@il.ibm.com

Abstract

Continuous Data Protection (CDP) is a recent storage technology which enables reverting the state of the storage to previous points in time. We propose four alternative architectures for supporting CDP in a storage controller, and compare them analytically with respect to both write performance and space usage overheads. We describe exactly how factors such as the degree of protection granularity (continuous or at fixed intervals) and the temporal distance distribution of the given workload affect these overheads. Our model allows predicting the CDP overheads for arbitrary workloads and concluding the best architecture for a given scenario. Our analysis is verified by running a prototype CDP enabled block device on both synthetic and traced workloads and comparing the outcome with our analysis. Our work is the first to consider how performance is affected by varying the degree of protection granularity, both analytically and empirically. In addition it is the first to precisely quantify the natural connection between CDP overheads and a workload's temporal locality. We show that one of the architectures we considered is superior for workloads exhibiting high temporal locality w.r.t. granularity, whereas another of the architectures is superior for workloads exhibiting low temporal locality w.r.t. granularity. We analyze two specific workloads, an OLTP workload and a file server workload, and show which CDP architecture is superior for each workload at which granularities.

1 Introduction

Continuous Data Protection (CDP) [13] is a new paradigm in backup and recovery, where the history of writes to storage is continuously captured, thereby allowing the storage state to be potentially reverted to any previous point in time. Typically the amount of history stored is limited by the operator, either in temporal terms using a *CDP window* (e.g. 2 days) or in terms of the amount of additional storage available for history data. CDP can be provided by different entities in the I/O path such as the host being protected (by the filesystem [28, 27, 24] or the Logical Volume Manager (LVM)), a SAN appliance [7], a SAN switch or the block storage controller. We focus on CDP enabling a block storage controller although parts of our work may be applicable elsewhere. We focus on the block level since this is typically the lowest common denominator of real world heterogeneous applications. We focus on the controller setting because it allows a wider range of architectures and has potential for performance and resource usage benefits. For example reverts do not generate network I/O, and both device I/O and space can potentially be reduced.

Advanced functions such as (writable) point-in-time copies and remote replication have already been introduced to storage controllers. The advent of these and other features has resulted in a growing divergence between the notion of logical and physical volumes. Moreover, log structured file systems [25] and log structured arrays (a similar notion realized in a controller setting) [22] have proposed exploiting this notion improve file system/controller performance for write requests by turning logically random write requests to physically sequential ones. CDP is an additional technology that furthers this trend, by introducing time as an additional dimension that can be virtualized. In this paper we study some of the associated performance tradeoffs.

Enabling the controller to save the history of host writes typically requires that the writes are duplicated and split. We present four different architectures for controller-based CDP that are differentiated by the point in time at which the writes are split: on receipt of the write by the controller, on destage of the write from the controller cache, on overwrite of a block by a newer version or never (avoiding duplication/splitting).

There is an ongoing debate [1] as to whether *continuous* protection (aka every write protection) provides much benefit to customers compared to coarser protection granularities. As a result some products going under the banner of CDP provide continuous (aka every write) protection, while other products are said to support *near* CDP, with protection granularities ranging from minutes to hours. CDP at whatever granularity does not come for free in either performance (impact on I/O throughput and latency) or disk space. We provide an exact analysis of the way the degree of protection (from granularity of every write to arbitrarily large fixed intervals) affects performance and space overheads.

When measuring performance, we focus on the count of additional device I/O's incurred by write requests in regular (non revert) scenarios for keeping track of prior versions and recording them in the CDP history. We focus on write requests since this is the main CDP overhead in good-path scenarios for 3 out of 4 of our architectures, where read performance is equivalent to that of a regular volume. Performance immediately after a revert depends on the data structures chosen to represent the CDP history and is the same across the architectures we compare since they share a common infrastructure for storing CDP history. Note that for appliance based block CDP, duplicating and splitting writes cannot be done at destage or overwrite time. Therefore our comparison of architectures also sheds light on inherent potential differences between appliance and controller based CDP.

Apart from logging write traffic, a controller offering a high-end CDP feature set must provide for fast revert of a production LUN to a prior point in time, be able to quickly export writable historical versions of the production volume, to perform automatic space reclamation of the older historical data etc. Although our proposed architectures all support these features, in this paper we do not describe the details and focus on the overheads CDP incurs when providing basic block I/O functionality.

We analyze our CDP architectures to accurately predict both the count of additional I/O's incurred by write requests and the space consumption overhead of each architecture. We describe exactly how factors such as the degree of protection granularity (continuous or at fixed intervals) and the temporal distance distribution of the given workload affect these overheads. Our work precisely quantifies the natural connection between CDP overheads and a workload's temporal locality. It allows predicting the CDP overheads for arbitrary workloads and concluding the best architecture for a given scenario.

We implemented a prototype CDP enabled block device and use it to validate our analysis against implementations of the architectures. We compare the cost of the architectures on real-world filesystem traces and synthetic OLTP traces, as we vary the protection granularity, and we conclude the best CDP architectures for these scenarios.

The outline of the paper is as follows. In section 2 we describe architectures for implementing CDP in a storage controller and in section 3 we characterize their performance analytically. Section 4 presents an evaluation of the performance of the proposed architectures on synthetic and real-life traced workloads. Section 5 reviews related work, and in section 6 we present our conclusions.

2 Architectures

2.1 Controller Background

Modern storage controllers typically contain a combination of processor complexes, read and fast write cache, host and device adapters, and physical storage devices. All components are typically paired so that there is no single point of failure [18, 17, 16]. Our figures describe a configuration having two *nodes*, where each one has one processor complex, one or more host adapters, one or more device adapters and one read and fast write cache, and each node owns a set of volumes (logical units (LUNs)) of the physical storage. To implement the fast write cache a subset of the total cache memory is backed by non volatile storage (NVS) on the opposite node. On node failure, volume ownership is transferred to the opposite node.

Cache (whether backed by NVS or not) is typically divided into units called *pages*, whereas disk is divided into units called *blocks* - pages/blocks are the smallest units of memory/disk allocation respectively. Stage and destage operations typically operate on an *extent*, which is a set of consecutive blocks. In the context of this paper, we concentrate on the case where the extent size is fixed and is equal to the page size. All architectures we discuss can be generalized to deal with the case that the extent size is some multiple of the page size although this requires delving into many details which need to be addressed for all architectures, and this is orthogonal to the main ideas we want to develop in the paper.

2.2 Integrating CDP into a Controller

We assume some mechanism for mapping between logical and physical addresses [1] where there is not a simple a priori relationship between them. Thin provisioning is another controller feature which can benefit from such a mapping. Such a mechanism must allow the cache to stage/destage from a physical address which differs from the logical address of the request. One way to do this is to allow the cache to invoke *callbacks*, so that a logical to physical map (LPMap) can be accessed either before or after stage and destage operations. In the case of CDP, timestamps will play a role in the LPMap.

We should distinguish between volumes that are *directly addressible*, and those which are *mapped* since they require a structure such as an LPMap in order to access them. Mapped volumes may be stored in individual physical volumes or alternatively several of them can be stored together in a larger physical volume which serves as a storage pool. Mapped volumes may sometimes be *hidden* from the user, as we will see for some of the architectures.

The LPMap structure supports the following API:

insert insert a mapping from a particular logical address to physical address, to be labeled by the current timestamp

lookup look up the current physical address corresponding to a particular logical address

revert revert the LPMap structure to a previous point in time

All architectures we consider will use the same LPMap representation.

2.3 Architectural Design Points

There are several factors which determine the CDP architecture. One factor is whether the current version of a volume is stored together with the historical data or separately from it. Storing current and historical data together results in the *logging* architecture. Note that this is not an option with host and network based CDP, where writes are duplicated and split at the host or at the network.

Since good sequential read performance is likely to be essential for most workloads, we consider alternative architectures containing 2 volumes - a directly addressible volume to hold the current version, and a hidden, mapped volume to contain historical data. In order to restore good sequential read performance, reverting such a volume to a previous point in time now inherently requires significant I/O - each changed extent needs to be physically copied from the history store to the current store. This I/O activity can be done in the background while the history store is used to respond to read requests as needed. This is similar to a background copy feature which accompanies some point-in-time copy implementations [10], and we expect the duration of such a background copy and the degree of performance degradation to be similar.

Assuming this separation between current and historical data, an important factor is when the duplication and splitting of I/O's is done. Splitting at write time leads to the *SplitStream* architecture, splitting at destage time leads to the *SplitDownStream* architecture, and splitting before overwrite time leads to the *Checkpointing* architecture. The corresponding location of the splitting would be above the cache, below the cache and at the storage respectively.

Note that our figures depict the current and history volumes as owned by opposite nodes, even though this is not necessarily the case. Moreover, in our figures we depict the CDP architectures as implemented on the previously described controller hardware. The figures might look slightly different if we were to design each particular CDP architecture from scratch with its own dedicated hardware.

A need may arise for *multi-version cache* support, namely the cache may need the ability to hold many versions of the same page. The CDP granularity requested by the user determines when data may be overwritten, for example every write granularity means that all versions must be retained and none overwritten. In this case, if the cache cannot hold multiple versions of a page, then an incoming write can force a destage of an existing modified cache entry. This needs to be taken into account since it can have affect the latency of write requests. How to best implement a multi-version cache is outside the scope of this paper.

2.4 The Logging Architecture

The logging architecture is the simplest - the entire history of writes to a volume is stored in a mapped volume which is not hidden from the user. Figure 1 depicts the write flow for the logging architecture. Note that each write request incurs at most 1 user data device I/O. If the CDP granularity is coarser than the rate of writes to a particular logical address, then there is potential to avoid this I/O if the logical address is cached.

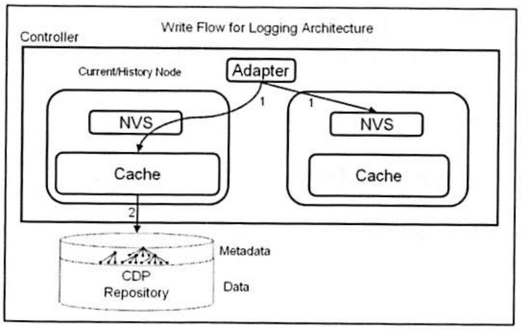


Figure 1: Logging Architecture Write Flow - each write request incurs at most 1 user data device I/O.

Both stages and destages need to access the LPMap structure. This approach is good for a write dominated workload, since not only do we avoid additional device I/O as a result of duplicating or copying from current volumes to historical volumes, we also have an opportunity to covert random writes into sequential ones, as done in [25]. An additional benefit is that with the right data structures for representing CDP history, revert can be done with a small constant overhead although the details are out of scope for this paper. Note that both reads and writes may require accessing the LPMap structures, although this does not always require additional device I/O, if these structures are intelligently cached. More importantly, the cost of accessing the LPMap is similar for all the architectures we discuss.

The important downside of this architecture is that one is likely to forfeit good sequential read performance, mainly because of the difficulty of sequential layout of extents on disk, and also because of access to the LPMap meta data. Although the layout is dependent on the implementation of the LPMap, which is out of the scope of this paper, guaranteeing good sequential read performance for this type of architecture is a difficult research problem [25]. Our simpler approach for achieving good sequential read performance is to separate current and historical data. Sequential read performance of historical data is not critical since it is only read immediately after a revert.

An additional issue with the logging architecture is that it is not straight forward to CDP enable an existing volume. Building the entire LPMap structure up front is not feasible, although if we choose to build it on demand, we may wait indefinitely until we can recycle the space of the original volume.

Note that the logging architecture is appropriate for a logging workload, since read access is only needed in error scenarios.

2.5 The SplitStream Architecture

Duplicating and splitting data above the cache leads to the *SplitStream* architecture. One copy of the data is sent to a *current store* - a directly addressable volume which holds the current version, and an additional copy is sent to a *history store* - a hidden, mapped volume, which in this case contains both current and previous versions of the data.

Figure 2 depicts the write flow for the SplitStream architecture. Note that each write request incurs at most 2 user data I/Os, one to each of the current and history stores. Just as for regular volumes, the cache can potentially save a current store I/O for those logical addresses that are repeatedly written to while they are still cache resident, and this effect is independent from the CDP granularity. If the CDP granularity is coarser than the rate of writes to a particular logical address, then there is potential to save a second I/O if that logical address is cached at the history store. Compared to the logging architecture, we gain good sequential read performance at the expense of incurring additional I/Os on write requests, and using more resources such as disk space and cache memory.

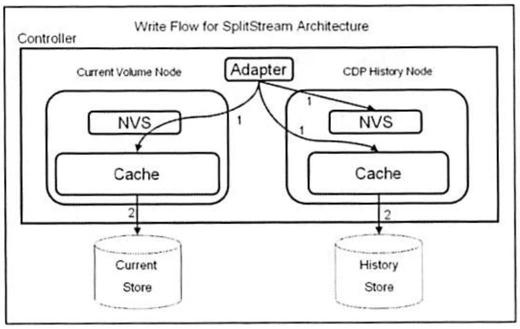


Figure 2: SplitStream Architecture Write Flow - Writes are split above the cache, and each write request incurs up to 2 user data device I/O's.

Note that in this case the cache can manage each volume separately, so a multi-versioned cache may not be a necessity. The current store does not need to deal with historical data, so can be cached as a regular volume, while the history store will not service reads, so caching simply serves as a means to buffer write access. Note that for coarse granularity the history store cache also serves to reduce device I/O for those logical addresses that are written to frequently. In the SplitStream architecture, the same data may appear twice in cache which inflates memory requirements, although the relative sizes of the current and history caches and the cache replacement algorithm could be tailored to take this into account.

2.6 The SplitDownStream Architecture

In the SplitDownStream architecture, the duplicating and splitting of data occurs under the cache, at destage time, instead of above the cache as is the case for SplitStream. In other respects the architectures are identical. This allows cache pages to be shared across current and historical volumes, thereby conserving memory resources. However, since the cache functions both to serve read requests and to buffer access to the history store, a versioned cache is needed to avoid latency issues.

Figure 3 depicts the write flow for the SplitDownStream architecture. Note that if we ignore cache effects which are different in the two architectures, SplitDownStream is identical to SplitStream in the number of device I/O's incurred by a write request.

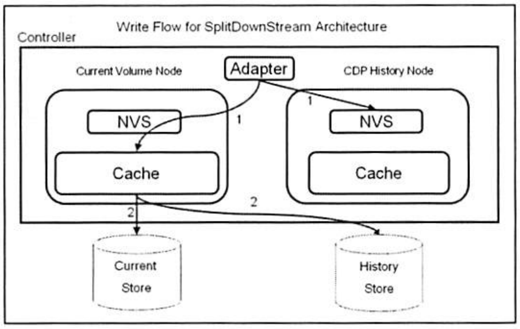


Figure 3: The SplitDownStream Architecture Write Flow - writes are split below the cache and each write request incurs up to 2 user data device I/O's.

2.7 The Checkpointing Architecture

Similar to the SplitStream and SplitDownStream architectures, the Checkpointing architecture has a directly addressible current store and a hidden mapped history store. However, in this case the current version exists only at the current store, while the history store contains only previous versions of the data. At destage time, before overwriting an extent at the current store, we first check whether it needs to be retained in the history store. This depends on the CDP granularity - every write granularity mandates that all versions need to be retained, whereas a coarse granularity requires very few versions to be retained. If needed, the extent is copied to the history store before being overwritten by the destage. Note that copying an extent from the current store to the history store requires 2 device I/O's - one to stage to the history store cache, and another to destage to the history store. Thus in total we may incur up to 3 device I/O's per write request. Figure 4 depicts the write flow for the Checkpointing architecture. If we do not use NVS at the history store, all 3 device I/Os are synchronous to the destage operation, although not to the write request itself, since we cannot destage until the previous version of the extent is safely on disk at the history store. Therefore to provide good latency a versioning cache is essential.

Of the total 3 possible device I/O's per write request, 2 of these (those for copying from the current to the history store) are not at all influenced by caching, and are completely determined by the CDP granularity. The other I/O (for destaging to the current store) can be avoided by caching if the particular version can be discarded according to the CDP granularity.

We point out that the checkpointing architecture is essentially an extension of the popular copy on destage technique used to implement Point In Time (PIT) copies [12] to the CDP context.

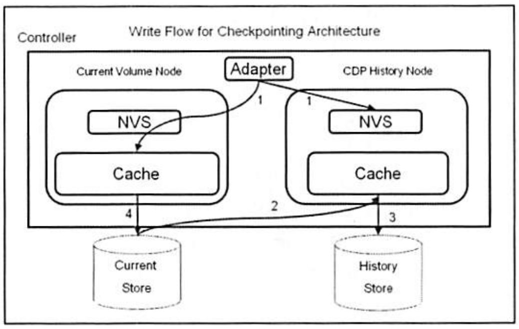


Figure 4: The Checkpointing Architecture Write Flow - the previous version is copied to the history store before being overwritten. Each write request incurs up to 3 user data device I/O's.

3 Analysis

In this section we analyze both the number of device I/O's incurred by write requests and the space overheads of the various CDP architectures, as a function of the CDP granularity.

3.1 Preliminaries

The time dimension can be divided into a set of fixed length **granularity windows** of size g. The requirement is to retain the last write in each granularity window. The smaller the value of g, the finer the granularity. When g is arbitrarily small, this results in every write CDP, whereas when g is arbitrarily large, this results in a regular volume with no CDP protection.

We define a write to a particular logical address to be **retained** if it is the last in its granularity window at that address. Fine granularity gives a large proportion of retained writes, while the opposite is true for coarse granularity. We use r to denote the proportion retained writes ($0 \leq r \leq 1$). For example, every write granularity gives $r = 1$ and as we increase g so that the granularity becomes coarser, r approaches 0. [2]

With respect to a fixed granularity, increasing the temporal distances between writes decreases the probability of a write to be retained. For a trace of a given workload, the **temporal distance distribution** of the workload is a function T, where for a given temporal distance t, $T(t)$ is the fraction of writes [3] with distance $\leq t$ to the subsequent write, so $0 \leq T(t) \leq 1$. There are a number of examples of the use of temporal distance distribution graphs in the literature [26, 32].

We use c to take write caching into account, where c is the proportion of writes which are *evicted* from cache before being overwritten ($0 \leq c \leq 1$). $c = 1$ means no absorption of writes by cache, whereas $c = 0$ means all writes can potentially be absorbed. Note that $1 - c$ is the write cache hit rate. Like the cache hit rate, c depends both on attributes of the write cache namely its size and replacement algorithm, as well as on attributes of the workload, namely its temporal locality and the distribution of its writes across logical addresses. The distribution of writes has a clear effect since repeated writes to a single address require less cache space than spreading the same number of writes across many addresses [4].

We define a write to be **incurred** if it is either retained, or evicted from cache before being overwritten, or both. In the logging architecture, the incurred writes are those which lead to a device I/O. We use d to denote the proportion of incurred writes under the logging architecture ($0 \leq d \leq 1$). It is important to note that in general $d \neq r + c - rc$ since being evicted from cache and being retained are not necessarily independent events.

3.2 Device I/O's Incurred by Write Requests

For a given r, c and d, figure 5 summarizes the I/O's incurred for the various architectures per write request. Note that both r and d are dependent on the granularity g. In section 3.4 we show how to derive r, and in section 3.5 we show how to derive d.

For example, if we have a workload with flat temporal locality of 0.5 seconds, then a once per second granularity gives $r = 0.5$. If we assume no cache then the expected cost of checkpointing per write request is $rp + (1 - r)p'$, where p is the cost for a retained write and p' is the cost for a non retained write in the checkpointing architecture. This equals $3r + (1 - r) = 2r + 1$. However for SplitStream and SplitDownStream the cost is always 2. The turning point is at $r = 0.5$, a larger value of r will give an advantage to Split(Down)Stream, whereas a smaller value will by advantageous to Checkpointing. r is determined by the relationship between the temporal locality of a given workload and the CDP granularity chosen by the user. Note that r is oblivious to the distribution of the writes, since the same fraction of retained writes can be as easily obtained with the same number of writes to a single logical address as to 1000 addresses, only the relationship of the temporal distance of the writes and the granularity chosen is important.

Caching does have an effect, and it should be clear from figure 5 that Split(Down)Stream can take better advantage of the cache than Checkpointing. Checkpointing can utilize cache to save non retained I/O's at the current store, but Split(Down)Stream can do the same and more. At an unrealistic extreme of infinite cache, SplitStream reduces I/O's to r per write request, the bare minimum (and equal to the Logging architecture). Here the expected cost of checkpointing per write request is $3r$, since all retained writes reach the history store via the current store. Note that SplitDownStream is slightly different from SplitStream in its behavior on retained I/O's at the current store. Since we destage cache entries to the current and history stores together, all retained writes reach the current store. In summary, increasing the size of the cache reduces c, and the SplitStream architecture gains most benefit from this, closely followed by the SplitDownStream architecture.

Unlike the case for regular volumes ($r = 0$), as r increases, the benefit which can be obtained by increasing the cache size becomes more limited. For example, for the extreme case of $r = 1$ e.g. every write granularity, the I/O cost for Checkpointing per write is 3 independent of the cache size, while for SplitDownStream it is 2. There is no crossover in this case and SplitDownStream always dominates. At the opposite extreme of a regular (non CDP enabled) volume ($r = 0$), the cost of checkpointing is c, whereas for Split(Down)Stream it is $2c$, giving a crossover at $c = 0$. This means that Checkpointing dominates Split(Down)Stream for all values $c > 0$ e.g. for an arbitrarily large cache. The reason is that Split(Down)Stream always splits and duplicate all writes, so it costs more than a regular volume when cache entries are destaged. Checkpointing, however, behaves like a regular volume in this case.

In general the crossover between Checkpointing and SplitStream is obtained at $c = 2r$, where if $c > 2r$ then Checkpointing dominates. This means that the proportion of writes evicted from cache before being overwritten needs to be at least twice the proportion of retained writes in order for Checkpointing to dominate. Since this is impossible when $r > 0.5$, Split(Down)Stream always dominates in that scenario.

3.3 Space Overhead

The fraction of retained writes r also determines the space needed to hold the CDP history, so there is a natural relationship between performance in terms of I/O counts incurred by writes and CDP space overhead. Figure 6 summarizes the space overhead for the various architectures. w denotes the number of writes within a given CDP window, a denotes the size of the addressable storage, and f denotes the size of the storage actually addressed during the CDP window. Since the Logging architecture is space efficient, it can have a lower space cost than a regular volume, whereas the cost of Split(Down)Stream equals the cost of a regular volume and a Logging volume combined. Checkpointing saves some space overhead since the current version is not stored in the history store.

		Logging	Checkpointing	SStream	SDownStream	Crossover
retained I/O's	current store		1	c	1	
	history store		2	1	1	
	sub total ($r = 1$)	1	3	$1 + c$	2	never
evicted I/O's	current store		c	c	c	
	history store		0	c	c	
	sub total ($r = 0$)	c	c	$2c$	$2c$	$c = 0$
total I/O's	**no cache** ($c = 1$)	1	$2r + 1$	2	2	$r = 0.5$
	infinite cache ($c = 0$)	r	$3r$	r	$2r$	$r = 0$
	arbitrary cache	d	$d + 2r$	$d + c$	$2d$	$c = 2r$

Figure 5: A table of device I/O's incurred per write request in terms of r, the proportion of retained writes and c, the proportion of writes which are destaged before being overwritten, and d, the proportion of writes resulting in a device I/O under the logging architecure. We show the analytical crossover point between Checkpointing and SplitStream Architectures.

Architecture	Space Overhead
Regular	a
Logging	rw
Split(Down)Stream	$a + rw$
Checkpointing	$a + rw - f$

Figure 6: Space overhead of CDP architectures in terms of r, w, a and f, where w is the total number of writes, r is the proportion of retained writes, a is the size of the addressable storage and f is the size of the storage actually addressed.

3.4 Retained Writes as a Function of the Granularity

We already mentioned that the proportion of retained writes r depends on the relationship between the temporal distance between writes and the granularity. Given a temporal distance distribution T, we show how to express r in terms of T and the granularity g. This allows us to infer properties of both device I/O counts incurred by writes as well as the space overhead of CDP for any given workload according to its temporal distance distribution.

We divide the time dimension into a set of fixed length granularity windows of size g. We assume that the first window starts at a uniform position between 0 and g. Let $R(g) = E(r(g))$ where E denotes expectation.

Claim 3.1

$$R(g) = 1 - \frac{1}{g} \int_{t=0}^{g} T(t)dt$$

Proof 3.1 *Let $w_1, \ldots, w_n$ be the set of writes in our given trace, and let a_i be the temporal distance between w_i and the subsequent write at the same logical address. Let X_i be the random variable that is 1 iff w_i is last in its granularity window, and so is a retained write. We have that $R(g) = E(\frac{1}{n}\sum_{i=1}^{n} X_i) = \frac{1}{n}\sum_{i=1}^{n} E(X_i)$.*

Now, $X_i = 1$ and w_i is a retained write iff it is within distance a_i from the end of the window. Since we place the start of the first granularity window at a uniform offset, we have that $E(X_i) = min(\frac{a_i}{g}, 1)$. Thus, $E(X_i) = \frac{1}{g}\int_{t=0}^{g} f_i(t)$, where $f_i(t) = 1$ if $t \leq a_i$ and is 0 otherwise.

Altogether, $R(g) = \frac{1}{n}\sum_{i=1}^{n} E(X_i) = \frac{1}{g}\int_{t=0}^{g} \frac{1}{n}\sum_{i=1}^{n} f_i(t)$ and $\frac{1}{n}\sum_{i=1}^{n} f_i(t) = 1 - T(t)$.

3.5 I/O Counts as a Function of the Granularity

Given a trace's temporal distance distribution T, we show how to express d as a function of g, assuming a certain write cache hit rate $(1 - c)$ and under a certain model of cache replacement. In order to simplify our analysis we will assume a Least Recently Written (LRW) replacement policy for the write cache, and will assume that the I/O rate is uniform. This means that writes wait in cache in a FIFO queue to be destaged, and it follows that the time interval writes wait from their last access until they are evicted is roughly constant for all writes. Given a cache hit rate $(1 - c)$, we define the *cache window* s to be the time interval that satisfies $c = 1 - T(s)$ and therefore is a result of this hit rate under our model.

Recall that d is the proportion of incurred device I/O's in the logging architecture. A write can result in a device I/O either because it is retained, or it is evicted from cache before being overwritten, or both. Recall that in general $d \neq r + c - rc$ since being evicted from cache and being retained are not necessarily independent events.

We show how to express d as a function of the granularity g and the cache window size s, where $c = 1 - T(s)$:

Claim 3.2 *Let R be as defined in claim 3.1.*

$$D(g,s) = \begin{cases} \frac{s}{g}R(s) + c(1-\frac{s}{g}) & \textit{if } s < g \\ R(g) & \textit{otherwise} \end{cases}$$

Proof 3.2 *Let $w_1, \ldots, w_n$ be the set of writes in our given trace, and let a_i be the temporal distance between w_i and the subsequent write at the same logical address. Let X_i be the random variable that is 1 iff w_i is an incurred write. A write w_i is incurred if it is either last in its granularity window or $a_i > s$ which causes the write to be evicted from the cache. If $s \geq g$, then $X_i = 1$ iff w_i is a retained write, because $a_i > s$ implies $a_i > g$ so all evicted writes are also retained. Therefore in this case $D(s,g) = R(g)$.*

Assuming $s < g$, we have $D(g,s) = E(\frac{1}{n}\sum_{i=1}^{n} X_i) = \frac{1}{n}\sum_{i=1}^{n} E(X_i)$. Since we place the start of the first granularity window at a uniform offset, we have:

$$E(X_i) = \begin{cases} \min(\frac{a_i}{g}, 1) & \textit{if } s > a_i \\ 1 & \textit{otherwise} \end{cases}$$

We define a function $f_i(t) = 1$ if $t \leq a_i$ and is 0 otherwise, and a function $\bar{f}_i(t) = f_i(t)$ if $t \leq s$ and $\bar{f}_i(t) = f_i(s)$ otherwise. Thus, we get $E(X_i) = \frac{1}{g}\int_{t=0}^{g} \bar{f}_i(t)$.

Altogether we get: $D(g,s) = \frac{1}{n}\sum_{i=1}^{n} E(X_i) = \frac{1}{g}\int_{t=0}^{g} \frac{1}{n}\sum_{i=1}^{n} \bar{f}_i(t) = \frac{1}{g}[\int_{t=0}^{s} \frac{1}{n}\sum_{i=1}^{n} f_i(t) + \int_{t=s}^{g} \frac{1}{n}\sum_{i=1}^{n} f_i(s)]$. Since we know that for any $t \in [0,g]$ it holds that $\frac{1}{n}\sum_{i=1}^{n} f_i(t) = 1 - T(t)$ we get $D(g,s) = \frac{1}{g}[\int_{t=0}^{s}(1-T(t)) + \int_{t=s}^{g}(1-T(s))] = \frac{1}{g}[sR(s) + (g-s)(1-T(s))]$.

This result can be used to calculate the I/O counts for the other architectures. Note that once the $s = g$, further increasing the cache size will not affect d and so will not provide additional benefit. This means that for fine granularity, cache size does not play an important role.

3.6 Discussion

We chose the LRW caching policy because it is simple to analyze and to implement. The result is a baseline comparison between the various architectures. Other caching policies such as WOW [15] could be considered and possibly adapted to the CDP context - this is a topic for further work. In practice the cache management policies implemented by modern storage controllers are complex and involve techniques such as prefetching and write coalescing. We did not introduce these aspects to our model however in order to keep it simple.

Our analysis applies to a segregated fast write cache, and an important topic of further work is to generalize it to the non segregated case which models the implementation of some storage controllers.

We already mentioned that a versioned cache is needed to avoid latency issues for all architectures except for SplitStream. Assuming a versioned cache, then some of the I/O's incurred by a write request are synchronous to the *destage* operation, but not to the write operation, unless the destage is synchronous to the write (such as when the write cache is full). Even though I/O's which are synchronous to a destage may not affect latency, they limit the freedom of the controller and may effect throughput. These are topics for further work.

Since there is an order of magnitude difference between random and sequential I/O bandwidth, the degree of sequentiality of the incurred I/O's also needs to be taken into account. For both sequential and random reads, Split(Down)Stream and Checkpointing architectures behave similarly. Regarding random writes, both architectures have the potential to convert random I/O to sequential I/O at the history store. However, for sequential write workloads Checkpointing seems to have a disadvantage, since on destage previous versions may need to be copied to the history store synchronously to the destage, which interferes with the sequential I/O flow to disk. This can be somewhat offset by optimizations such as proactively copying many adjacent logical addresses to the history store together. A more detailed analysis of effect of the sequentiality of workloads on the performance of the various architectures, and an empirical evaluation, is outside the scope of this paper and is a topic for further work.

It would be interesting to do a bottleneck analysis of the various architectures, although this is beyond the scope of our work. One point to consider is the timing of the incurred device I/O's and the resulting effect on the back-end interconnect. In the Checkpointing architecture, we may see bursts of activity once a CDP granularity window completes, whereas in the other architectures the additional load is more evenly spread over time.

4 Performance

In this section we analyze our CDP architectures in the context of both synthetic and real life workloads. We analyze the properties of the workloads that affect both CDP performance and space usage, as well as empirically measuring the performance of the various architectures using a prototype CDP enabled block device.

4.1 Experimental Setup

To evaluate the CDP architectures we presented in section 2, we implemented a prototype stand alone network storage server with CDP support. The prototype was written in C under Linux and offers a block storage I/O interface via the NBD protocol [5], and can also be run against trace files containing timestamped I/O's. The prototype has a HTTP management interface which allows reverting the storage to previous points in time. The CDP history structures were implemented using the B-Tree support in the Berkeley DB database library [6]. Our prototype has been tested extensively using a python test suite and has also been used to mount file systems.

Our prototype emulates a storage controller's cache with a LRU replacement policy for reads and LRW (least recently written) policy for writes. In a typical storage controller, dirty data pages are battery backed. In the controllers we model, for cost reasons, the number of these pages is limited to a small fraction of the total pages in the cache [17]. In our experiments we limited dirty pages to occupy at most 3% of the total cache size, and we varied the total cache size in our experiments. Based upon our experience, a ratio close to 3% is often seen in systems which have segregated fast write caches. We chose a page size and extent size of 4KB, and varied the granularity from every write granularity, across a range of granularity values.

4.2 Workloads

4.2.1 The SPC-1 Benchmark

Storage Performance Council's SPC-1 [9] is a synthetic storage-subsystem performance benchmark. It works by subjecting the storage subsystem to an I/O workload designed to mimic realistic workloads found in typical business critical application such as OLTP systems and mail server applications. SPC-1 has gained some industry acceptance and storage vendors such as Sun, IBM, HP, Dell, LSI-Logic, Fujitsu, StorageTek and 3PARData among others have submitted results for their storage controllers [8]. The benchmark has been shown to provide a realistic pattern of I/O work [20] and has recently been used by the research community [21, 15].

We compared the CDP architectures on workloads similar to ones generated by SPC-1. We used an earlier prototype implementation of the SPC-1 benchmark that we refer to as SPC-1 Like. The choice of a synthetic workload enabled us to monitor the effect of modifying workload parameters which is important for reaching an understanding of the behavior of the CDP architectures. The SPC-1 Like prototype was modified to generate a timestamped trace file instead of actually submitting the I/O requests. All trace files generated were 1 hour long.

A central concept in SPC-1 is the Business Scaling Unit (BSU). BSUs are the benchmark representation of the user population's I/O activity. Each BSU represents the aggregate I/O load created by a specified number of users who collectively generate up to 50 I/O's per second. SPC-1 can be scaled by increasing or decreasing the number of BSUs.

SPC-1 divides the backend storage capacity into so-called Application Storage Units (ASUs). Three ASUs are defined: ASU-1 representing a "Data Store", ASU-2 representing a "User Store" and ASU-3 representing a "Log/Sequential Write". Storage is divided between the ASUs as follows: 45% is assigned to ASU-1, 45% to ASU-2 and the remaining 10% is assigned to ASU-3. The generated workload is divided between the ASUs as follows: 59.6% of the generated I/Os are to ASU-1, 12.3% are to ASU-2 and 28.1% to ASU-3. Finally, another attribute of the SPC-1 workload is that all I/O's are 4KB aligned.

4.2.2 cello99 traces

cello99 is a well known block level disk I/O trace taken from the cello server over a one year period in 1999. cello is the workgroup file server for the storage systems group at HP labs and the workload is typical of a research group, including software development, trace analysis and simulation. At that time, cello was a K570 class machine (4 cpus) running HP-UX 10.20, with about 2GB of main memory. We used a trace of the first hour of 3/3/1999.

4.3 Workload Analysis

4.3.1 Temporal Distance Distribution

According to our analysis in sections 3, the temporal distance distribution is a crucial property of a workload which influences both performance of the various CDP architectures in terms of I/O counts and the predicted space usage. We observed that this distribution for SPC-1 Like traces is determined by the ratio between the number of BSUs (the load level) and the size of the target ASU storage. We obtained the same distribution when the number of BSUs and ASUs is varied according to the same ratio. We obtained a set of SPC-1 Like traces with different distributions of overwrite delays by varying this ratio. Increasing the number of BSUs while keeping the storage size constant means that more activity takes place in a given unit of time, and this decreases the expected overwrite delays the workload. We chose to vary the number of BSUs rather than the target storage size since this allows us to easily keep a fixed ratio between cache and storage size. For all trace files the total capacity of the ASUs is kept constant at 100GB. To modify temporal locality we varied the number of BSUs: 33, 50, 75 and

100. Our BSU/ASU ratio of 33 BSU/100 GB is comparable with certain SPC1 vendor submissions for high end storage controllers [8], therefore we expect the temporal distance distribution to be similar.

In all our measurements we ignored I/O's to ASU-3 since it represents a purely sequential-write workload (a log) and we wanted to avoid skewing our results according to this. Such a workload is characterized by very large overwrite delays, and a logging or special purpose architecture would be most suitable for providing CDP functionality. Ideally, one could choose a CDP architecture per protected volume.

Figure 7 shows the temporal distance distribution in the SPC-1 Like and cello99 traces. Note that for the SPC-1 Like traces, as the number of BSUs is increased, the average temporal distance decreases, which indicates that for a given granularity there is more potential for I/O and space savings. Note that the temporal distances exhibited by the cello99 are much shorter than those for the SPC-1 Like traces, and this leads to an expected difference in behavior of the CDP architectures.

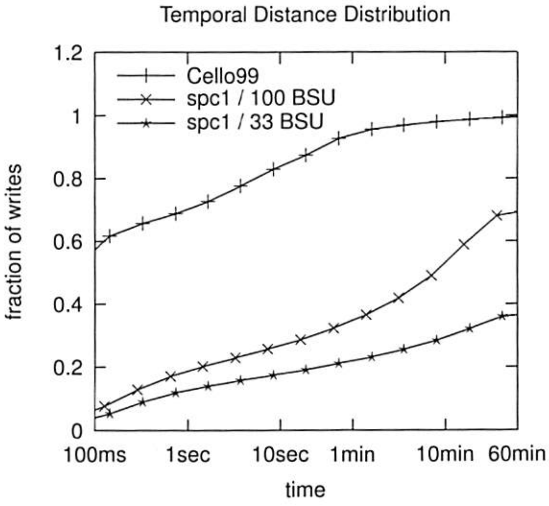


Figure 7: Temporal Distance Distribution of cello99 and SPC-1 Like workloads. For a given interval d the graph plots the fraction of writes with temporal distance $\leq d$ to the subsequent write.

4.3.2 Space Overhead and Retained Writes

Figure 8 shows the space overhead of the CDP Logging architecture as a function of granularity. The storage overhead of every write (EW) granularity is normalized to 1, and increasing the granularity reduces the space overhead. [5] Because of the relatively small temporal distances in the cello99 trace, considerable savings are possible at very fine granularities in the order of seconds. On the other hand, the relatively large temporal distances in the SPC-1 traces means that space savings are obtained only with granularities which are larger by several orders of magnitude. As shown in figure 6, the space overhead of the logging architecture equals rw, so our normalized graph is a graph of r, the proportion of retained writes, as a function of granularity. The relationship of the space overhead of the other architectures with granularity is similar, as can be derived from figure 6. We also calculated the expected proportion of retained writes analytically according to the formula from claim 3.1 and the given temporal distance distributions for the cello and SPC1 Like traces. Figure 8 compares our analytical results with our empirical ones, and we see an extremely close match, validating the correctness of our analysis.

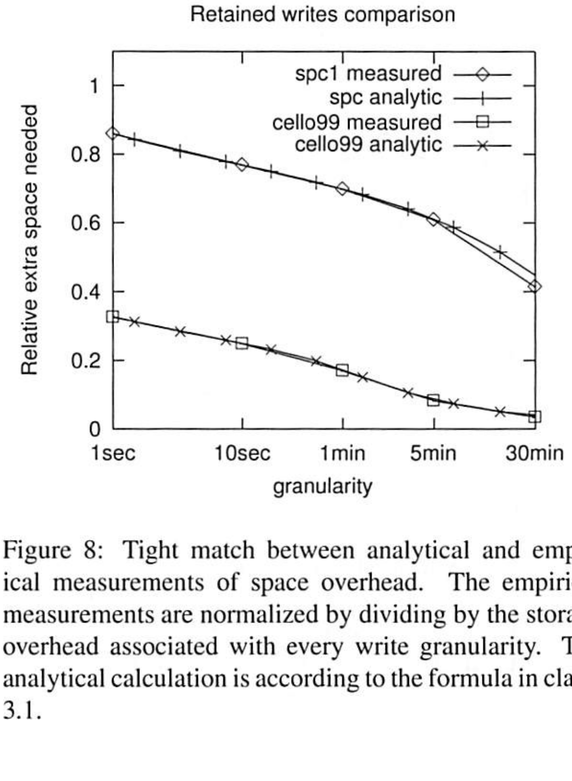


Figure 8: Tight match between analytical and empirical measurements of space overhead. The empirical measurements are normalized by dividing by the storage overhead associated with every write granularity. The analytical calculation is according to the formula in claim 3.1.

4.3.3 I/O Counts Incurred by Write Requests

We ran our prototype implementations on the Logging, Checkpointing and SplitDownStream architectures using the SPC1 Like trace with a 100BSU/100GB ratio and 4GB cache, and counted the number of I/O's incurred by write requests (see figure 9). All I/O counts are normalized according to the total number of extents written in the trace. At every write (EW) granularity, the architectures are close to a 1:2:3 ratio as expected according to our analysis. At a granularity coarser than 5 mins the Logging Architecture is close to the I/O counts of a regular volume. As the granularity becomes coarser, there is a very gradual improvement (note the logarithmic scale of the granularity axis), and SplitDownStream dominates Checkpointing for granularities up to 30 mins, at which point there is a crossover. The gradual improvement is a result of a low proportion of writes with temporal distances up to 5 minutes, as shown in figure 7. In figure 10, we perform the same experiment on our SPC1 Like trace with a 33BSU/100GB ratio and 4GB cache. Because of the reduced temporal locality, SplitDownStream dominates Checkpointing for 60 minute granularities and beyond.

In figure 11, we plot the empirical results we obtained for the 100BSU trace against the calculated analytic results for the same trace. As can be seen there is a close match, validating our analysis.

In figure 12, we performed a similar experiment using the cello99 trace with a cache size of 64MB. As for the SPC1 Like trace, at every write (EW) granularity the architectures are close to a 1:2:3 ratio, although the higher temporal locality of this trace results in a very fast decline of I/O counts for the checkpointing architecture with an increase in granularity, and it dominates SplitDownStream at 5 minute and coarser granularities. At a granularity of 60 mins, I/O counts for checkpointing approach those Logging and of a regular volume.

In figure 13, we plot the empirical results we obtained for the cello99 trace against the calculated analytic results for the same trace. As can be seen there is a close match, validating our analysis.

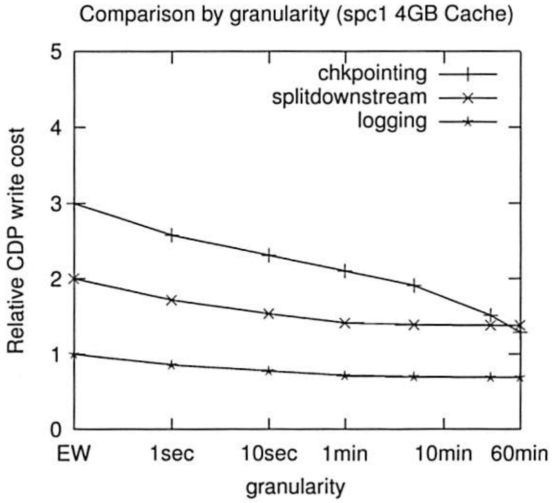


Figure 9: A comparison of I/O counts incurred by write requests for the CDP architectures as a function of granularity, for the SPC1 Like trace with 100BSU/100GB. There is a crossover point after 30mins where Checkpointing overtakes SplitDownStream - note the logarithmic scale of the X axis.

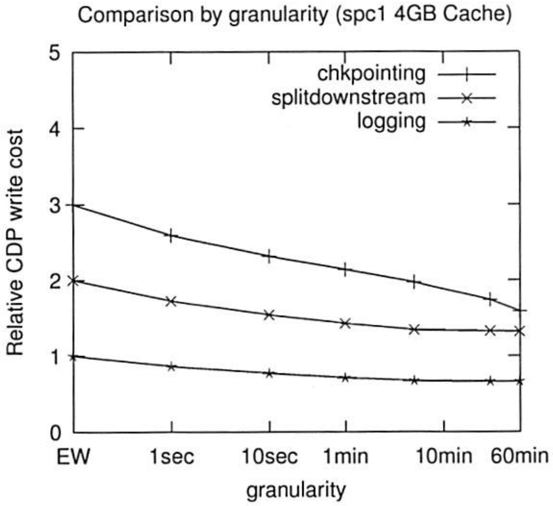


Figure 10: A comparison of I/O counts incurred by write requests for the CDP architectures as a function of granularity, for the SPC1 Like trace with 33BSU/100GB. Since this trace exhibits less temporal locality, the crossover point of figure 9 has moved to beyond 1 hour, in favor of SplitDownStream.

4.3.4 The Effect of Write Cache on I/O Counts

It is clear that increasing the write cache size reduces the I/O counts for any architecture, assuming the CDP granularity allows this. According to our analysis, the Checkpointing architecture is more limited than SplitDownStream in its ability to utilize write cache to reduce I/O counts. In figure 14, we compare the I/O counts of the various architectures as the cache size is increased, for the cello99 trace with 1 second granularity. The performance of all 3 architectures enjoy an increase in cache size, with the SplitDownStream architecture best able to utilize the additional cache space. In general we expect an increase in cache size to push the crossover point between SplitDownStream and Checkpointing to coarser granularities in favor of SplitDownStream - whereas the crossover point for 64MB in figure 12 is less than 10 minutes, for approx. 100MB it is exactly 10 minutes, and for 512MB it is greater that 10 minutes. Note that in this case a large increase in cache size makes only a small difference to the crossover granularity. Also note that once the cache size reaches approximately 256MB, a further increase in cache size does not further reduce the CDP write cost. For the 256MB cache size, we found s to be close to 10 minutes, which confirms our analysis from section 3.5 that predicted this to happen once $s = g$.

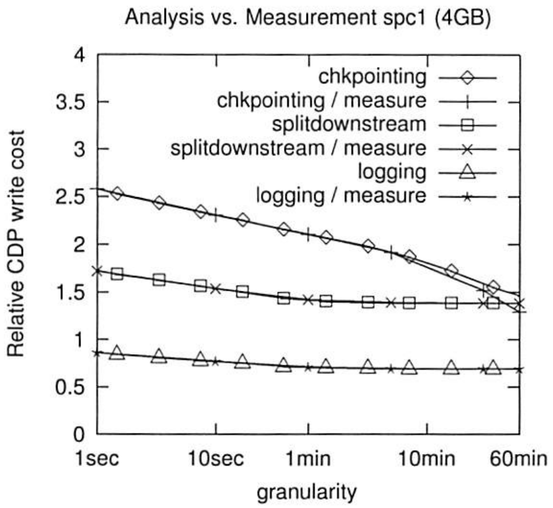


Figure 11: Tight match between analytical and empirical measurements of I/O counts for the SPC1 Like 100BSU/100GB trace. The analytical calculation is according to the analysis developed in 3.

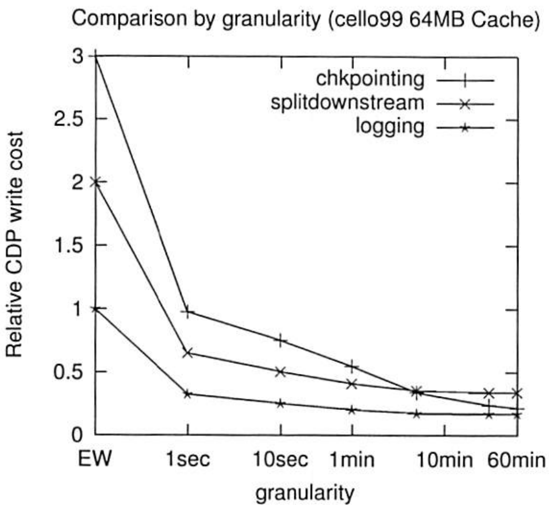


Figure 12: A comparison of I/O counts incurred by write requests for the CDP architectures as a function of granularity, for cello99 with 64MB cache. Checkpointing dominates SplitDownStream at 5 minute and coarser granularities. The reason for the very different crossover point is the much larger degree of temporal locality in the cello99 trace.

5 Related Work

Although CDP has been gaining momentum in the industry and various products are available [7, 2, 4, 3] we are not aware of an enterprise-class storage controller with CDP support available as a product. There has been some research concerned with implementing CDP [23, 31, 14] but to the best of our knowledge we are the first to describe CDP architectures suitable for implementation in a high-end storage controller.

There has been some work examining every-write block based CDP with a focus on reducing space overhead [23, 31]. The paper describing the Peabody system makes a case for content-based coalescing of sectors: for the workloads they investigated, up to 84% of the written sectors have identical contents to sectors that were written previously and so could potentially be stored just once [23]. The paper describing the Trap-array system proposes to reduce the size of the CDP history by storing the compressed result of the XOR of each block with its previous version, instead of storing the block data. They find that it is common for only a small segment of a block to change between versions so that a XOR between the two versions yields a block containing mostly zeros the compresses well. Their results show up to two orders of magnitude space overhead reduction. The resulting cost is that retrieving a block version requires time proportional to the number of versions between the target version and the current version [31].

These results seem promising and the ideas presented in these papers can be considered complementary to our work. It may be interesting to investigate one of the proposed schemes as function of a variable protection granularity as neither paper does so.

The Clotho system is a Linux block device driver that supports creating an unlimited amount of read-only snapshots (versions) [14]. Since snapshot creation is efficient, frequent snapshots are feasible. Clotho is similar to the logging architecture in that read access to the current version requires metadata lookups. Unlike the logging architecture in Clotho access to newer versions is more efficient than access to historical versions. Similarly to the Trap-array system, a form of differential compression of extents with their previous versions is supported however its benefits on real-world workloads is not quantified.

Point-in-time volume snapshots [12] are a common feature supported by storage controllers, LVMs, filesystems and NAS boxes. We believe the results of our analysis of the checkpointing architecture are relevant for analysis of the overhead of periodic point-in-time snapshot support when this is implemented using COW (copy on write): on overwrite of the block on the production volume, the previous version of the block is copied to the snapshot volume. We presented results and analysis which provide insight into when COW-based architectures should be used and when alternatives should be considered as protection granularity and workload vary.

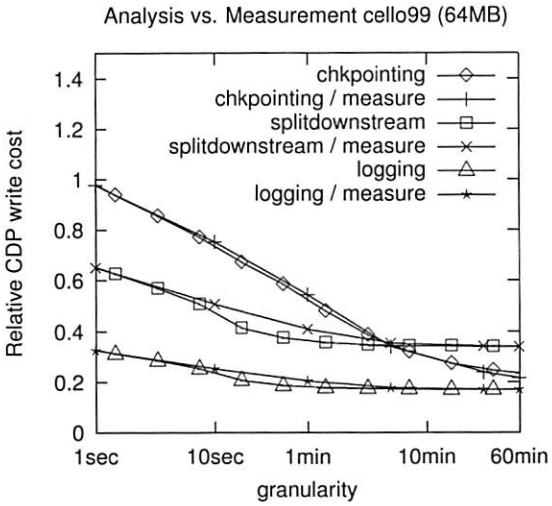


Figure 13: Tight match between analytical and empirical measurements of I/O counts for cello99 trace. The analytical calculation is according to the analysis developed in 2

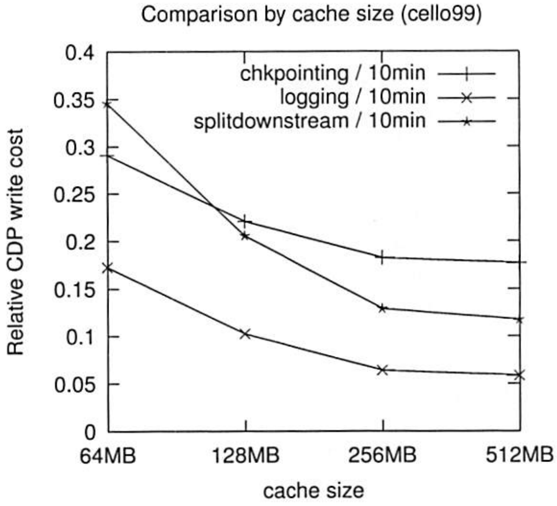


Figure 14: Write cache size versus relative CDP cost in terms of I/O's incurred by write requests for the 1 hour trace of the cello99 workload with a granularity of 10 minutes. As the cache size is increased, the cost reduces faster for the SplitDownStream architecture than for Checkpointing, and SplitDownStream dominates Checkpointing for cache sizes over 100MB. Note the exponential scale of the X-axis.

We are not aware of prior work which examined the issue from this perspective.

Network Appliance NAS filers and the Sun ZFS [11] filesystem support snapshotting of volumes by organizing the filesystem as a tree with user-data at the leaf level and meta-data held in internal nodes [19]. Each snapshot consists of a separate tree which may share subtrees with other snapshots. Creating a snapshot volume is done by creating a new tree root node which points to the same children as the original volume's root, so that the two volumes are represented by overlapping trees. Once a snapshot is created on a volume, any blocks which it shares with the snapshot (initially all blocks) cannot be updated in place, and all writes to them must be redirected. The first write to a block after a snapshot causes an entire tree path of meta-data to be allocated and copied, and linked to from the snapshot volume's tree root. Compared to in-place updates of meta-data, this approach seems to inherently require writing much more meta-data, especially for frequent snapshots. Some of this cost is balanced by delaying writes and then performing them in large sequential batches, similar to LFS [25]. However at high protection granularity it is not clear how competitive this architecture is. An investigation of the performance of this architecture as function of workload and protection granularity may be an interesting further work item.

Other work examined two options for implementing point-in-time snapshots, referred to as: COW and 'redirect on write' (ROW) [30]. The performance of a volume which has a single snapshot on it is examined and the two options are compared. No consideration is given to the cost related to periodically creating a new point-in-time copy. Very roughly the redirect-on-write architecture can be compared to the logging architecture while the COW architecture can be compared to our checkpointing architecture. The I/O cost per write for ROW is 1 I/O and, similarly to the case for logging, there is an extra meta-data lookup per read as well as a loss of spacial locality. The intuition that for write-dominated workloads ROW has an advantage while for read-dominated ones COW is advantageous is experimentally validated on various workloads. The impact of block size (minimal unit of copying to snapshot) on performance of the architectures is also investigated. Briefly, smaller block sizes benefit writes since less space is wasted while it hurts reads because of more fragmentation.

Versioned file systems [28, 27, 24, 29] keep track of updates to files and enable access to historical versions. The unit of protection in these file systems is a file - the user may access a historical version of a specific file - while in block-based CDP the unit of protection is an entire LUN. Some of this work discusses the overheads of meta-data related to versioning [24, 29] but our focus has been on the overheads associated with user data. The basic approach of investigating write related overheads as a function of protection granularity seems applicable to versioned file systems as well. One difference is that some versioned file systems support protection granularities that are not a constant amount of time, e.g. all the changes made to a file between the time it was opened by an application and the time it was closed may be considered to belong to a single version, whatever the amount of time this happens to take. This is similar to so called 'event-based' CDP where the user/application marks events of interest rather than deciding beforehand about protection granularity. Extending our model to handle this may be a further work item.

6 Conclusions

We proposed CDP architectures suitable for integration into storage controller, and analyzed them both analytically and empirically. Our analysis predicts the cost of each architecture in terms of write-related I/O and space overheads, and our empirical results confirm the analysis. Our work is the first to consider and accurately describe the effects of varying the CDP granularity. We show that one of the critical factors affecting both write I/O and space overheads is the fraction of retained writes which is determined by the relationship between the temporal distance of writes and the granularity. Workloads exhibiting high temporal locality w.r.t. the granularity perform well under the checkpointing architecture, whereas workloads exhibiting low temporal locality w.r.t. the granularity perform well under the SplitDownStream architecture. We analyzed specific workloads and showed that for the SPC1 Like OLTP workloads, a SplitDownStream architecture is superior for granularities up to 1 hour. We also showed that the Checkpointing architecture is superior for a workgroup file server workload such as cello99 for granularities coarser than 5 minutes. Aside from CDP, our results can also shed light on the performance overheads of common implementations of point-in-time copy in terms of the frequency of taking those copies.

6.1 Further Work

We do not claim definite conclusions regarding the cost of the CDP architectures for the general class of OLTP workloads or filesystem workloads. Our evaluation was not extensive enough to substantiate such claims. However we believe our results lay a foundation for a thorough investigation of real-world workloads.

A hybrid architecture which combines ideas from the Checkpointing architecture and the SplitDownStream architecture may offer the best of both: the behavior of SplitDownStream at higher granularities and the behavior of Checkpointing at lower granularities. Such an architecture is an interesting further work item.

Our evaluation did not consider I/O's related to metadata accesses to the LPMap CDP History structure. Also ignored in our evaluation is sequentiality of the write-related traffic. As described the checkpointing architecture requires 2 synchronous I/O's prior to cache destages of some pages. Evaluating the impact this has on performance (as well as attempting to overcome this limitation of the checkpointing architecture) is a further work item.

Acknowledgments

We would very much like to thank Uri Braun for his contribution at the early stages of our prototype, Chip Jarvis for discussions on versioning cache, Bruce McNutt for allowing us access to an earlier prototype implementation of the SPC-1 benchmark, HP for access to the cello99 traces, and Binny Gill and Dharmendra Modha for sending us their WOW prototype. Many thanks also to Kalman Meth, Sivan Tal, and the anonymous reviewers of the FAST program committee whose valuable feedback helped improve the paper.

References

[1] Continuous Data Protection: A Market Update, Byte and Switch Insider Report, July 2006. http://www.byteandswitch.com/insider.

[2] EMC Recover Point. http://www.emc.com/.

[3] FalconStor CDP. http://www.falconstor.com/.

[4] Mendocino Software, http://www.mendocinosoft.com/.

[5] Network Block Device. http://nbd.sourceforge.net/.

[6] Oracle Berkeley DB. http://www.oracle.com/database/berkeley-db/index.html.

[7] Revivio Inc. http://www.revivio.com/.

[8] Storage Performance Council, SPC-1 Benchmark Results. http://www.storageperformance.org/results.

[9] Storage Performance Council SPC-1 Specification. http://www.storageperformance.org/specs.

[10] The IBM TotalStorage DS8000 Series: Concepts and Architecture. IBM Redbook, 2005, http://www.redbooks.ibm.com/.

[11] ZFS: The last word in file systems. http://www.sun.com/2004-0914/feature/.

[12] A. Azagury, M. E. Factor, J. Satran, and W. Micka. Point-in-time copy: Yesterday, today and tomorrow. In *Proceedings of IEEE/NASA Conf. Mass Storage Systems*, pages 259–270, 2002.

[13] J. Damoulakis. Continuous protection. *Storage, June 2004*, 3(4):33–39, 2004.

[14] M. Flouris and A. Bilas. Clotho: Transparent data versioning at the block I/O level. In *IEEE Symposium on Mass Storage Systems*, 2004.

[15] B. Gill and D. S. Modha. WOW: Wise ordering for writes combining spatial and temporal locality in non-volatile caches. In *Proceedings of USENIX File and Storage Technologies*, 2005.

[16] J. S. Glider, C. F. Fuente, and W. J. Scales. The software architecture of a SAN storage control system. *IBM Systems Journal*, 42(2):232–249, 2003.

[17] M. Hartung. IBM TotalStorage Enterprise Storage Server: A designer's view. *IBM Systems Journal*, 42(2):383–396, 2003.

[18] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach; second edition*. Morgan Kaufmann, 1996.

[19] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the Winter'94 USENIX Technical Conference*, pages 235–246, 1994.

[20] B. McNutt and S. A. Johnson. A standard test of I/O cache. In *Proc. of the Computer Measurements Group Conference*, 2001.

[21] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of USENIX File and Storage Technologies, San Francisco, CA.*, 2003.

[22] J. Menon. A performance comparison of RAID-5 and log-structured arrays. In *Fourth IEEE Symposium on High-Performance Distributed Computing*, 1995.

[23] C. B. Morrey III and D. Grunwald. Peabody: The time travelling disk. In *IEEE Symposium on Mass Storage Systems*, pages 241–253, 2003.

[24] Z. Peterson and R. Burns. Ext3cow: a time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, May 2005.

[25] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured filesystem. *ACM Transactions on Computer Systems*, pages 26–52, 1992.

[26] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proceedings of the Winter USENIX Conference*, pages 405–420, 1993.

[27] D. J. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Otir. Deciding when to forget in the elephant file system. In *SOSP99, Symposium on Operating Systems Principles*, 1999.

[28] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmers workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 25-34, 1985.

[29] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of USENIX File And Storage Technologies*. USENIX, 2003.

[30] W. Xiao, Y. Liu, Q. K. Yang, J. Ren, and C. Xie. Implementation and performance evaluation of two snapshot methods on iSCSI target stores. In *Proceedings of IEEE/NASA Conf. Mass Storage Systems*, 2006.

[31] Q. Yang, W. Xiao, and J. Ren. TRAP-Array: A disk array architecture providing timely recovery to any point-in-time. In *Proceedings of International Symposium on Computer Architecture*, 2006.

[32] Y. Zhou and J. F. Philbin. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of USENIX Annual Tech. Conf., Boston, MA*, pages 91–104, 2001.

Notes

[1] With the advent of RAID and volume virtualization, these are not necessarily true physical addresses.

[2] To calculate r, we only consider completed granularity windows, so that it becomes evident which writes are retained permanently. For "infinite" granularity, there is only one granularity window but it is never completed.

[3] We only consider those writes which have a subsequent write.

[4] The distribution of writes may not be as important if a versioning cache is used.

[5] Note that one cannot properly normalize according to the case of a regular volume (no CDP) because there are no retained writes, and there is no relationship between the size of the current stores in the various workloads.

Jumbo Store: Providing Efficient Incremental Upload and Versioning for a Utility Rendering Service

Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes

HP Laboratories

{kave.eshghi,mark.lillibridge,lawrence.wilcock,guillaume.belrose,rycharde.hawkes}
@hp.com

Abstract

We have developed a new storage system called the Jumbo Store (JS) based on encoding directory tree snapshots as graphs called HDAGs whose nodes are small variable-length chunks of data and whose edges are hash pointers. We store or transmit each node only once and encode using landmark-based chunking plus some new tricks. This leads to very efficient incremental upload and storage of successive snapshots: we report compression factors over 16x for real data; a comparison shows that our incremental upload sends only 1/5 as much data as Rsync.

To demonstrate the utility of the Jumbo Store, we have integrated it into HP Labs' prototype Utility Rendering Service (URS), which accepts rendering data in the form of directory tree snapshots from small teams of animators, renders one or more requested frames using a processor farm, and then makes the rendered frames available for download. Efficient incremental upload is crucial to the URS's usability and responsiveness because of the teams' slow Internet connections. We report on the JS's performance during a major field test of the URS where the URS was offered to 11 groups of animators for 10 months during an animation showcase to create high-quality short animations.

1 Introduction

Utility Computing describes the notion that computing resources can be offered over the Internet on a commodity basis by large providers, and purchased on-demand as required, rather like gas, electricity, or water. The widespread belief is that computation services can be offered to end users at lower cost because of the economies of scale of the provider, and because end users pay only for the resources used at any moment in time.

Utility services are utility computing systems that offer the functionality of one or more software applications rather than raw processing or storage resources. Possible utility services include finite element analysis, data mining, geological modeling, protein folding, and animation rendering. An important class of utility service, which we call batch services, primarily processes batch jobs where each job involves performing a well-defined set of computations on supplied data then returning the results of the computations. The data for a job may be large and complicated, consisting of many files carefully arranged in a file hierarchy—the animation models for rendering a movie short can require gigabytes of data and thousands of files.

Providing batch services to individual consumers or small and medium businesses under these circumstances is difficult because the slow Internet connections typical of these users make moving large amounts of data to the servers very time-consuming: uploading the animation models for a movie short over a typical ADSL line with 256 Kbits/s maximum upload bandwidth can take over 17 hours. (Downloading of results is usually less problematic because these connections offer much greater download bandwidths.)

We believe this problem can be solved in practice for many batch services if incremental uploading can be used since new jobs often use data only slightly different from previous jobs. For example, movie development, like computer program development, involves testing a series of successive animation models, each building on the previous one. To spare users the difficult and error-prone process of selecting which files need to be uploaded, the incremental uploading process needs to be automatic.

We have developed a new storage system, the Jumbo Store (JS), that stores Hash-Based Directed Acyclic Graphs (HDAGs). Unlike normal graphs, HDAG nodes refer to other nodes by their hash rather than by their location in memory. HDAGs are a generalization of Merkle trees [20] where each node is stored only once but may have multiple parents. Filesystem snapshots are stored on a Jumbo Store server by encoding them as a giant HDAG wherein each directory and file is represented by a node and each file's contents is encoded as a series of variable-size chunk nodes produced by landmark-based chunking (cf. LBFS [21]). Because each node is stored only once, stored snapshots are automatically highly compressed as redundancy both within and across snapshots is eliminated.

The Jumbo Store provides a very efficient form of incremental upload: the HDAG of the new snapshot is generated on the client and only the nodes the server does not already have are sent; the presence of nodes on the server is determined by querying by node hash. By taking advantage of the properties of HDAGs, we can do substantially less than one query per node. We show that the JS incremental upload facility is substantially faster than its obvious alternative, Rsync [26], for movie animation models.

As well as being fast, the upload protocol requires no client state and is fault tolerant: errors are detected and corrected, and a restarted upload following a client crash will not start from scratch, but make use of the portions of the directory tree that have already been transmitted. The protocol also provides very strong guarantees of correctness and completeness when it finishes.

To demonstrate the utility of the Jumbo Store, we have integrated it into a prototype Utility Rendering Service (URS) [17] developed by HP Labs, which performs the complex calculations required to create a 3D animated movie. The URS is a batch service which accepts rendering data in the form of directory tree snapshots from small teams of animators, renders one or more requested frames using a processor farm, and then makes the rendered frames available for download.

The URS research team involved over 30 people, including developers and quality assurance specialists. It is designed for use by real users and so has to be user friendly and easy to integrate into the customer computing infrastructure, with a high level of security, quality of service, and availability. To provide performance and security isolation, one instance of the URS is run for each animator team. Each URS instance uses one JS server to store that team's uploaded animation model snapshots. Each service instance may have multiple snapshots, allowing animator teams to have multiple jobs running or scheduled at the same time. Because of JS's storage compression, we can allow a large number of snapshots inexpensively.

To test the URS, it was deployed for each of 11 small teams of animators as part of an animation showcase called SE3D ("seed") [27], which ran for a period of 10 months. The URS gave the animators access to a large pool of computing resources, allowing them to create high quality animated movie shorts. The system was highly instrumented and the participants were interviewed before and afterwards. We report extensively in the second half of this paper on the JS's excellent performance during SE3D. As far as we know, this trial is the only substantial test of incremental upload for utility services.

The remainder of this paper is organized as follows: in the next section we describe the design and implementation of the Jumbo Store. In Section 3, we briefly describe the URS and how it uses the JS. In Section 4, we describe the results of the SE3D trial. In Section 5, we compare JS to Rsync using data from SE3D. In Section 6, we discuss the SE3D and Rsync comparison results. Finally, in the remaining sections we discuss related work (Section 7), future work (Section 8), and our conclusions (Section 9).

2 The Jumbo Store

The Jumbo Store (JS) is our new storage system, which stores named HDAGs—immutable data structures for representing hierarchical data—called *versions*. The JS is accessed via special JS clients. Although HDAGs can hold almost any kind of hierarchical data, we currently only provide a client that encodes snapshots of directory trees as HDAGs. This client allows uploading new snapshots of the machine it is running on, downloading existing snapshots to that machine, as well as other operations like listing and deleting versions. Figure 1 below shows the typical configuration used for incremental upload. A version can be created from the (recursive) contents of any client machine directory or from part of an existing version; in either case, files can be filtered out by pathname.

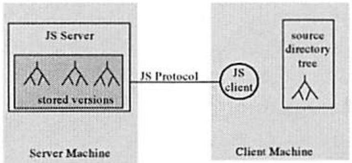
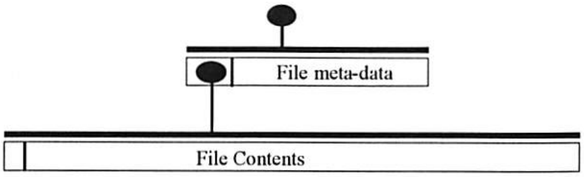


Figure 1: Incremental upload configuration

2.1 Hash-based directed acyclic graphs

An HDAG is a special kind of directed acyclic graph (DAG) whose nodes refer to other nodes by their hash rather than their location in memory. More precisely, an HDAG is a set of HDAG nodes where each HDAG node is the serialization of a data structure with two fields: the pointer field, which is a possibly empty array of hash pointers, and the data field, which is an application-defined byte array. A hash pointer is the cryptographic hash (e.g., MD5 or SHA1) of the corresponding child. Pictorially, we represent a hash pointer as a black dot that is connected to a solid bar above the node that is hashed. For example, a file can be represented using a two level HDAG:



The leaf node's data field contains the contents of the file and the root node's data field contains the file's meta-data. Using this representation, two files with the same data contents but different metadata (e.g., different names) will have different metadata nodes but share the same contents node: because nodes are referred to by hash, there can be only one node with a given list of children and data.

Continuing our example, we can extend our representation to arbitrary directory structures by representing each directory as a node whose data field contains that directory's metadata and whose children are the nodes representing the directory's members. Figure 2 below shows an example where the metadata nodes for ordinary files have been suppressed to save space; each grey box is a contents node.

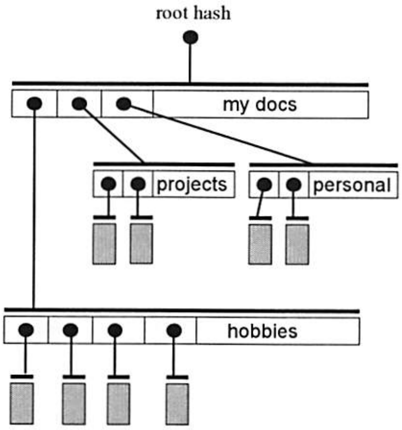


Figure 2: An HDAG representation of a directory tree

HDAGs are a generalization of Merkle trees [20]. They are in general not trees, but rather DAGs since one child can have multiple parents. Also unlike Merkle trees, their non-leaf nodes can contain data. Notice that even though a directory structure (modulo links) is a tree, its HDAG representations are often DAGs, since there are often files whose contents are duplicated in whole or in part (see chunking in Section 2.3). The duplicated files or chunks will result in two or more HDAG nodes pointing to the same shared node.

2.2 Properties of HDAGs

We say that an HDAG is *rooted* if and only if there is one node in that HDAG that is the ancestor of all the other nodes in the HDAG; we call such a node the HDAG's *root node* and its hash in turn the HDAG's *root hash*. An HDAG is *complete* if and only if every one of its nodes' children also belongs to that HDAG; that is, there are no 'dangling' pointers. Figure 2 above is an example of a rooted, complete HDAG.

HDAGs have a number of useful properties.

Automatically acyclic: Since creating an HDAG with a cycle in the parent-child relation amounts to solving equations of the form

$$H(H(x;d_2);d_1) = x$$

where H is the underlying cryptographic hash function, which we conjecture to be cryptographically hard, we think it is safe to assume that any set of HDAG nodes is cycle free. All of the HDAGs we generate are acyclic barring a hash collision and it seems extremely unlikely that a random error would corrupt one of our HDAG nodes, resulting in a cycle.

Unique root hash: given two rooted, complete (acyclic) HDAG's H_1 and H_2, they are the same if and only if their root hashes are the same. This is a generalization of the 'comparison by hash' technique with the same theoretical limitations [16]; in particular, this property relies on the assumption that finding collisions of the cryptographic hash function is effectively impossible. More precisely, it stems from the fact that a root hash is effectively a hash of the entire HDAG because it covers its direct children's hashes which in turn cover their children's hashes and so on. By induction, it is easy to prove that if H_1 and H_2 differ yet have the same root hash, there must exist at least two different nodes with the same hash.

Automatic self assembly: Because all the pointers in an HDAG are hashes, given an unordered set of HDAG nodes we can recreate the parent-child relationship between the nodes without any extra information. To do this, we first de-serialize the nodes to get access to the hash pointers. We then compute the hash of every node. Now we can match children with parents based on the equality of the hash pointer in the parent with the hash of the child.

Automatic structure sharing: Not just single nodes are automatically shared within and between HDAGs; sub- DAGs representing shared structure are as well. Consider Figure 3 below; it shows two snapshots of the same directory tree taken on adjacent days. Only one file (labeled old/new file) changed between the snapshots. Every node is shared between the two snapshot representations except the modified file's content node, its metadata node (not shown), and the nodes representing its ancestor directories. In general, changing one node of an HDAG changes all of that node's ancestor nodes because changing it changes its hash, which changes one of the hash pointers of its parent, which changes its parent's hash, which changes one of the hash pointers of its grandparent, and so on.

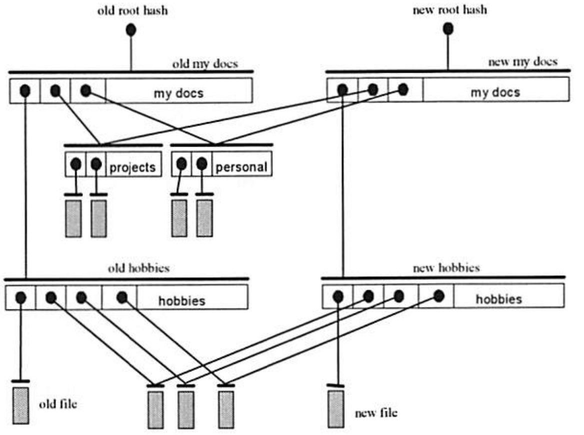


Figure 3: Structure sharing between HDAGs

2.3 Snapshot representation

The snapshot representation described in Section 2.1 has the major drawback that if even one byte of a file is changed, the resulting file's content node will be different and will need to be uploaded in its entirety. To avoid this problem, we break up files into, on average, 4 KB pieces via content-based chunking.

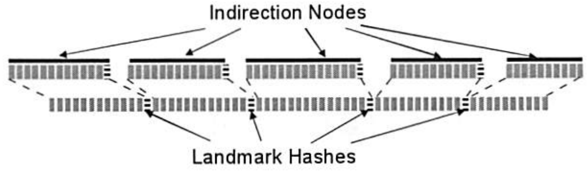
Content-based chunking breaks a file into a sequence of chunks based on local landmarks in the file so a local modification to the file does not change the relative position of chunk boundaries outside the modification point [21,22]. This is basically equivalent to breaking a text file into chunks at newlines but more general; editing one line leaves the others unchanged. If we used fixed size blocks instead of chunking, inserting or deleting in the middle of a file would shift all the block boundaries after the modification point, resulting in half of the file's nodes being changed instead of only one or two.

We use the two-threshold, two-divisor (TTTD) chunking algorithm [13], which is an improved variant we have developed of the standard sliding window algorithm. It produces chunks whose size has smaller variance; this is important because the expected size of the node changed by a randomly-located local change is proportional to the average chunk size plus the variance divided by the average chunk size. (Larger chunks are more likely to be affected.)

2.3.1 The chunk list

With chunking, we also need to represent the list of hashes of the chunks that make up a file. We could do this by having the file metadata node have the file's chunks as its children. However, the resulting metadata node can become quite large: since we currently use 17-byte long hashes (MD5 plus a one byte hash type), a 10 MB file with average chunk size of 4 KB has approximately 2,500 chunks so the list of chunk hashes alone would be 42 KB. Since the smallest shared unit can be one node, to maximize sharing it is essential to have a small average node size. With this representation, changing one byte of this file would require sending over 46 KB of data (1 chunk node and the metadata node).

We introduce the idea of chunking the chunk hash list itself to reduce the amount of chunk list data that needs to be uploaded when a large file is changed. We chunk a list of hashes similarly to file contents but always place the boundaries between hashes and determine landmarks by looking for hashes whose value = -1 mod *k* for a chosen value of *k*. We package up the resulting chunk hash list chunks as indirection nodes where each indirection node contains no data but has the corresponding chunk's hashes as its children:



We choose our chunk list chunking parameters so that indirection nodes will also be 4 KB on average in size; this corresponds to about 241 children. We use chunking rather than just dividing the list every *n* hashes so that inserting or deleting hashes does not shift the boundaries downstream from the change point. Thus, even if ten chunks are removed from the beginning of the file, the indirection nodes corresponding to the middle and end of the file are not affected.

This process replaces the original chunk list with a much smaller list of the hashes of the indirection nodes. The resulting list may still be too large so we repeat the process of adding a layer of indirection nodes until the resulting chunk list is smaller than a desired threshold, currently 2. Files containing no or only one chunk of data will have no indirection nodes. The final chunk list is used as the list of children for the file metadata node.

The result of this process is an HDAG at whose leaves are the chunks, and whose non-leaf nodes are the indirection nodes. This HDAG, in turn, is pointed to by the file metadata node. Thus, we use the chunking scheme and the indirect nodes as a natural extension of the HDAG representation of directory structures (see Figure *2*).

Under this representation, a 10 MB file has approximately 2,500 data chunks, 11 first level indirection blocks, one second level indirection block, and one file metadata node. The overhead of making a small change in this file ignoring the metadata node's contents and ancestors is the size of one chunk (~4 KB) plus the size of one first level indirect node (~4 KB) plus the size of the second indirect block (~17 bytes), which sums up to roughly 8 KB, which is much better than the 46 KB a flat representation would have required.

2.4 Efficient incremental upload and storage

Efficient incremental upload for snapshots can be described as follows: there is a site, the source, where an up-to-date copy of a directory structure exists, and another site, the target, where one or more older snapshots of the same directory structure exist. The connection between the two sites may be slow and unreliable. It is required to create a snapshot of the current contents of the source directory on the target, minimizing the transfer time and maximizing the reliability.

The properties of HDAGs make them ideal for use in implementing efficient and reliable transfers such as incremental upload: First, the automatic self-assembly property means that the HDAG nodes can be gathered from multiple sources (e.g., possibly stale caches), in any order. No matter where the nodes come from, there is only one way to put them together to get an HDAG.

Second, the unique root hash property lets us check when a transfer has successfully and correctly completed: if the target has a complete, rooted acyclic HDAG whose root hash is the same as that of the source HDAG then we have a strong guarantee that the HDAG at the target is identical to the HDAG at the source. Any extra received nodes not part of this HDAG (e.g., nodes corrupted in transit) may be discarded. If the HDAG is incomplete, it is easy to determine the hashes of missing nodes.

Third, the automatic structure sharing property ensures that many nodes will be shared between the source and target. Such nodes need not be transmitted if they can be determined to already be present on the target. This can be done by querying the existence of each source node at the target by hash (this uses much less space than sending the node itself). Fourth, by taking advantage of the unique root property, it is possible to query the existence of an entire sub-DAG with root hash *h* by sending a single hash, *h*: if the target replies that it has a complete sub-HDAG with root hash *h*, then it must have the same sub-DAG the source has; this 'compare by root' technique can be much more efficient than querying about individual nodes when a lot of structure is shared.

Combining these ideas, we get the following algorithm: The incremental upload algorithm runs on the client system. Let *H* be the complete HDAG representing the source directory structure. The client agent traverses *H* in some order and for each node *N* encountered, queries the remote server whether it has the complete DAG whose root is *N*. *N* is transmitted only if the answer is negative. If the answer is positive, then the children of *N* need not be traversed. The remote server replies with the hashes of the nodes it receives, allowing retransmission if needed. Once the client has finished traversing *H*, it tells the remote server to finalize the version using the HDAG with root hash *H*'s root hash.

There is a great deal of flexibility in the order in which the nodes of *H* are generated and traversed. In particular, we do not require that the whole of *H* be in memory at the same time. Moreover, there can be multiple threads working on different parts of *H* concurrently. Currently, to bound the client's RAM usage even though files may be arbitrarily large, we use compare by root only for DAGs representing files whose root hashes we already know from a previous upload; for all other nodes, we query existence individually. By default, we maintain a small cache on the client of previously uploaded normal files mapping their pathname plus modification time when last uploaded to the root hash of the DAG that represented them. If these files have the same modification time as the last time they were uploaded, we can avoid regenerating their representative DAGs if compare by root succeeds.

2.4.1 Efficiency and reliability

Our current algorithm is efficient: each untouched file requires one query for compare by root, each other node (e.g., directories and the nodes in changed files) requires one query for compare by hash, and each new node additionally must be transmitted. Because queries contain only a single hash, which is 240 times smaller than the average 4 KB node size we use, we effectively send only the parts of the HDAG that have been modified since the previous snapshot. By careful design of our snapshot representation (see Section 2.3), we have ensured that small local changes to the source directory structure change as few HDAG nodes as possible.

To minimize the latency of the query-response, queries and nodes are sent in one thread while the responses are processed asynchronously in another thread; we also batch messages to reduce overhead.

HDAG nodes are computed in parallel with querying/sending. Computing HDAGs is relatively fast: a 3.2 GHz Xeon Windows PC can scan, compute HDAG nodes, and count how many unique HDAG nodes there are in an in-cache (i.e., no disk I/O) filesystem tree that contains 64 directories, 423 files, and 220 MB of data at over 18 MB/s. Accordingly, in our experience HDAG computation time is normally dominated by transmit time (slow links) or client disk scan/read time (fast links).

Because the JS stores each node only once, these same properties allow us to store multiple successive snapshots of the same directory tree in very little space; in effect, storing another snapshot requires as much space as would be required to incrementally upload that snapshot.

What happens if something goes wrong during the upload process? If some nodes get corrupted in transit, then we will detect that by comparing the returned hashes, and the nodes will be re-sent. What if the upload process is interrupted for some reason? Let us say that 70% of the way through the transfer the client crashes. All we have to do is to start the upload process again from the beginning (no client state need be kept). Since we still have all the HDAG nodes that have already been transferred on the server, very quickly the client will reach the same point in the process where the previous transfer was interrupted, and continue from there. The only time lost is the time to scan the source directory and construct the HDAG again, which is a fraction of the transfer time. Because of the strength of the cryptographic hash we use and unique root hash property, we can be very sure if the transfer succeeds that no errors have been made.

2.5 Implementation

The JS server, about 13,000 lines of C++, runs on a single Windows or Linux machine and supports multiple concurrent client TCP connections. The basic JS client is a command-line program, about 15,000 lines of pure Java, which can run on any operating system that supports Java 1.4.

The Jumbo Store, unlike other content-addressable stores [5,24,29], is an *HDAG-aware store*. That is, in addition to operations to store and retrieve the basic unit of storage (the node for JS) by hash, the JS server supports operations on entire HDAGs. For example, it supports 'compare by root' queries ("do you have a complete HDAG with root hash *h*?"), "how big is the HDAG with root hash *h*?", and the deletion of entire HDAGs (really versions). The JS server does not interpret nodes' data fields and knows nothing of snapshots. The protocol the JS speaks has no connection-specific state and all messages are idempotent, allowing easy retransmission in case of lost messages or connections.

JS data is stored in a series of large data files on disk; an in-memory hash table indexes the nodes stored in the data files by their MD5 hash. A separate file for each version contains only that version's root hashes—partial versions may have multiple roots. To support deletion, the index also maintains a reference count for each node where each version root is considered a root for the purposes of reference counting. Occasionally a background process compacts data files by copying only the nodes with a nonzero reference count to a new file. This simple reference counting garbage collection scheme works well because HDAGs are acyclic.

Due to space limitations, we will not discuss downloading snapshots or the other operations the JS client supports further except to note that we use a sophisticated tree pre-fetching algorithm to avoid pipeline stalls during downloading.

3 The Utility Rendering Service

The Utility Rendering Service (URS) is a batch utility service that performs the calculations required to render a 3D animated movie. It gives animators access to a large pool of resources to perform the rendering, and allows them to purchase rendering resources when needed. Animation is an interesting domain in which to test technologies for Utility Services because of the natural cycles in demand for resources inherent in a typical movie production cycle.

The URS does not fundamentally change the way in which an animator works; they still use the tools they are familiar with. However, it does offer the potential for a more efficient and interactive style of work because animators have access to a more powerful set of resources than they could otherwise economically afford, allowing the visual quality settings to be turned up, and allowing the animator to be more experimental because the turnaround time for scenes is reduced.

The Utility Services model is particularly attractive for small animation organizations, because it allows them to acquire computing resources at short notice when needed, allowing individuals and small teams to dynamically form and take on projects that would otherwise not be possible if only in-house computing resources were used. Because of space limitations, we will concentrate here on only the aspects of the URS that are relevant to the use of the JS.

3.1 User model

Animators use a commercial content creation application called Maya® [3] to create the digital models that define their 3D animated movie, including the shape and movement of characters, backgrounds, and objects, and associated textures, lighting, and camera definitions. Maya uses over a dozen file formats including a variety of image formats (e.g., JPG and TIFF) and several proprietary formats; most of these are binary formats, although a few are ASCII (e.g., the MEL scripting language).

To interact with the URS, animators use a Java application called the URS Client, running on one or more of their computers. The URS Client allows users to upload input data, submit rendering jobs, monitor the progress of jobs, download rendered frames, and manage the data stored on the server.

We imagine a dynamic, competitive market for Utility Services, where customers may only subscribe to a service on demand and for limited periods, based on factors such as price and functionality. Accordingly, the barrier for successful subscription to, and use of, a service needs to be low. Towards this end, the URS client is written in pure Java for operating system portability, automatically works through firewalls, is easy to download, and is self updating.

The URS separates the tasks of uploading animation models, rendering models into frames, and downloading frames for viewing, allowing them to be performed independently and, in many cases, in parallel. Uploading input data (a directory tree specified by the user containing a consistent set of files that can be rendered) results in a new snapshot of the input data stored at a URS server; these snapshots are referred to as "versions" by the URS system. Versions remain until explicitly deleted by a user but are subject to an overall space quota. Note that the root of the input data directory tree can be changed each time a new version is created, so, unlike a source-code versioning system like CVS, the structure of the files and directories may change radically from one version to the next.

To render frames, an animator submits a new job request against a specific version, specifying the name of a scene file within that version and the frame numbers to compute. A job can be submitted against a version any time after its uploading has been initiated. Allowing multiple jobs per version and rendering multiple versions at the same time greatly increases flexibility. For example, an animator may wish to interactively make several changes to a character model and experiment with which looks best, and have the rendering service compute each possibility simultaneously.

Newly rendered frames are downloaded in the background by default as they become available. Alternatively, animators may explicitly request when and which frames should be downloaded.

3.2 Architecture

The overall architecture and data flow of a URS instance is shown in Figure 4 below. A server-side subsystem of URS, the Asset Store, manages the transfer and storage of the input and output data. The Asset Store consists of two processes (Asset Manager and Jumbo Store) and three internal storage areas, each with an associated storage quota that users must keep within.

The Version File Store stores the data managed by the JS server process; it contains in compressed form the available URS versions and possibly a partial version in the process of being uploaded. The Output Content Store stores the rendered frames generated by processing nodes.

The remaining storage area is the Version Cache (VC), which stores a subset of the versions held in the Jumbo Store in their fully expanded form, ready for use by the processing nodes. The VC is needed because the JS currently only supports uncompressing an entire snapshot at a time, a time-consuming operation, and there is not enough room to keep every version in expanded form.

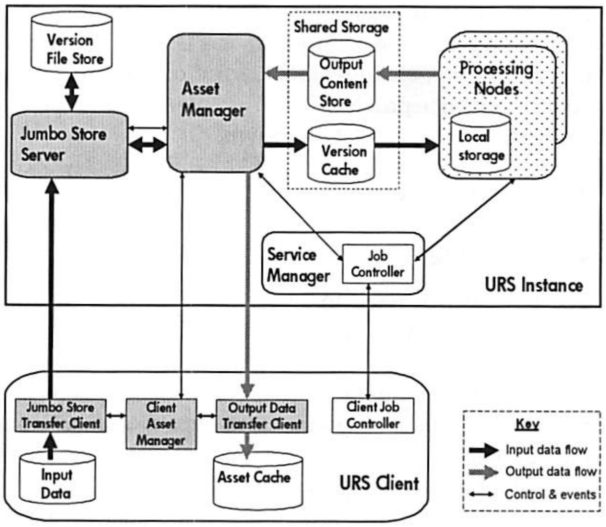


Figure 4: URS architecture and data flow

The lifecycle and state of input data versions is managed by the Asset Manager. Versions have a well-defined lifecycle, representing the stages of creation, transfer, archival to Jumbo Store, restore to VC, deletion from VC, and removal. Important changes to the Asset Manager state are held persistently in a database so that state can be fully recovered on service instance restart even after failure. Incomplete asynchronous operations on input data versions, such as upload, extraction, or deletion, are either cancelled or completed as appropriate. To keep the design simple, only one upload is permitted at a time per service instance.

3.3 Client-server communication

Communication between all components running in the URS Client and those in the URS is implemented over a single Secure Socket Layer (SSL) encrypted socket connection made from the client. This gives automatic client firewall traversal, the ability to easily terminate in a single operation on the server all interactions with a specific user, and, similarly, the ability to reestablish communication in the event of temporary connection failure with a single operation. However, the disadvantage is that all data, control, and event protocols must be multiplexed down a single channel.

All client-server communication, for data, control, and events, is implemented over a simple object passing and addressing abstraction called the Message Object Broker (MOB), which is layered above the SSL socket. The MOB allows serialized Java objects to be exchanged across the socket to named recipients on the remote side, and offers a variety of call semantics such as request-reply, buffered writes, and direct object passing. It also implements a simple keep-alive mechanism, shared by all protocols using the MOB to detect connection failures. The pure Java implementation strategy, and the use of serialized Java objects, did not prove to be a problem for acceptable performance of bulk data transport.

4 SE3D Results

4.1 Setting

The URS was offered to 11 small teams of animators during an animation showcase, called SE3D, to create high-quality short animations. The SE3D animation showcase was a unique experiment, conducted over a period of 10 months, giving new, creative talent from the animation industry access to a set of research technologies for Utility Services, together with a large pool of computer resources. The trial involved up to 120 dual 3 GHz Xeon processor servers, each with 4 GB RAM, and a total of 4 TB of storage. The URS server-side components, including the Jumbo Store servers, were deployed in a data centre in the US, while the animators were all located in the UK. Thus all data transfers had to traverse the public Internet over a transatlantic link.

There was considerable variation between teams in working methods, kinds of Internet connections, number of animators using the URS, how often and how many times they uploaded, how many client machines they used, how big their movie source was, and the like. Table 1 below summarizes each team's use of the URS upload facility; to preserve privacy we have assigned teams service instance numbers in order of increasing movie source size. Here, 'uploads' is the total number of uploads attempted by that instance and 'logged' is the number of those uploads for which we have correctly logged information—because JS was added to the URS after SE3D started and because some early bugs caused bad logging, we do not have useful information for some early transfers; in particular, we have no trustworthy data for instance 0 so it is omitted from the rest of this paper. The remaining two columns give the average version size (i.e., movie source size) and average number of files involved in the correctly logged uploads for that service. Note that size here refers to the size of the version on the client, not the amount actually transferred to or stored at the Jumbo Store.

service instance	uploads	logged	average size (MB)	average # files
0	43	0		
1	124	17	92.0	24.7
2	87	68	109.8	21.2
3	287	286	143.7	5025.5
4	217	122	342.7	145.0
5	379	263	351.4	76.4
6	32	29	352.3	91.4
7	229	209	360.1	99.8
8	55	42	1873.6	225.6
9	125	109	2498.5	773.6
10	202	169	3046.3	4709.3
avg	161.8	119.5	917.0	1119.3
all	1780	1314	859.3	1929.0

Table 1: Use of the upload facility

All but one service exploited the Jumbo Store's ability to hold multiple versions in order to render multiple versions at the same time. Although most services rendered a maximum of three or four versions simultaneously, two services rendered 7 and 10 respectively versions at the same time.

4.2 Reliability and robustness

Transferring gigabytes of data via TCP without higher level end-to-end checking and retransmission is problematic: given TCP's 16-bit checksum and assuming a 1% packet error rate and 1500 byte packets, we expect an undetected data corruption error to occur once every 9.2 GB of data. Indeed, the authors were unable to check out a 12 GB Subversion repository over the transatlantic cable due to repeated network errors and Subversion's inability to restart incomplete transfers where they left off. By contrast, our first attempt to copy the same data via JS worked perfectly.

When used independently, Jumbo Stores verify each received chunk using cryptographic checksums, requesting retransmission as needed, to handle transmission errors. They also reconnect transparently should a TCP connection be broken due to an error or a timeout. Accordingly, neither kind of error requires restarting an upload.

As incorporated into the URS, Jumbo Store traffic is sent over SSL using a supplied MOB connection. Because of SSL's cryptographic checksums, any data corruption results in a broken connection. Unfortunately, while the URS can automatically reestablish a new connection, it cannot do so in a manner transparent to the MOB's clients, which include the JS. It can, however, automatically restart at the beginning an upload aborted due to a broken connection. Because the JS upload protocol does not resend data already on the Jumbo Store, we quickly scan forward to the furthest point the upload previously reached.

During SE3D, there were 262 restarts, the vast majority of which (251) were for service instance 5, whose Internet connection appears to have been unreliable at times—1 transfer restarted 58 times before the user stopped it. Inspecting the logs shows that 91.7% of the uploads succeeded, 7.8% of the uploads were aborted by users before they completed, and 0.4% of the uploads failed due to URS problems unrelated to the JS. At most 12% of the user aborts can be attributed to frequent restarts. The remaining aborts are presumably due to users realizing they had made a mistake or wishing to upload instead an even newer version. If we count the later as successes, then the overall URS upload success rate exceeds 98.6%.

The only version data loss we suffered occurred early on due to a bug in the JS server's garbage collector. The bug was quickly fixed and we were able to recover much of the data from the URS Version Cache.

4.3 Compression

For the purposes of this and Section 4.4, we analyze only the 1092 uploads (83% of the correctly logged JS uploads) that succeeded, did not restart, and immediately follow a successful upload. This is necessary to ensure meaningful statistics; e.g., an aborted upload may have partially uploaded a snapshot, making the next upload seem artificially efficient.

Table 2 below shows average compression ratios (i.e., compressed size/uncompressed size) of various kinds for each of the instances (1-10), all the instances treated as a single service (all), and the average service instance average compression ratio (avg, the average of the individual instance numbers). The all numbers differ from the avg numbers because they more heavily weigh instances with large numbers of uploads/versions. We will quote both numbers as avg # (all #).

service instance	upload	within version	across versions	both
1	20%	39%	37%	16%
2	9.3%	48%	16%	10%
3	6.7%	69%	9.7%	6.8%
4	1.4%	41%	3.7%	1.7%
5	2.1%	30%	5.6%	2.1%
6	19%	81%	21%	17%
7	1.6%	34%	4.1%	2.0%
8	1.6%	75%	9.1%	7.2%
9	0.52%	28%	15%	3.8%
10	1.1%	36%	1.5%	1.0%
avg	6.3%	48%	12.3%	6.8%
all	3.5%	44%	7.3%	4.0%

Table 2: Various compression ratios

The upload column shows the average upload ratio of the actual number of data and metadata bytes uploaded over the total number of data bytes in the snapshot being uploaded. Thus, a conservative approximation of our upload compression ratio is 6.3% (3.5%); equivalently, our upload compression factor (1/ratio) is 16x (29x). While analyzing the logs, we discovered a performance bug: a second write of a block while its first write was still in progress could result in that block being transmitted twice. We conservatively estimate that had this bug been fixed beforehand, our upload compression would have instead been 5.5% (3.2%) or 18x (31x).

The 'within version' column shows the average version storage compression ratio under the restriction that no sharing is permitted between versions; the restriction is equivalent to requiring each version to be stored on a separate Jumbo Store by itself. These numbers—48% (44%) or 2.1x (2.3x)—are surprisingly good and indicate that movie sources are fairly redundant.

The 'across versions' column attempts to measure the degree of storage compression due to sharing between versions (of the same service instance) rather than within versions. It shows the average ratio of the additional storage required to store a new version on a Jumbo Store containing all surviving previous versions over the amount of storage required to store that version separately. Between version compression gives us 12.3% (7.3%) or 8.1x (14x). Note that the degree of storage compression possible due to sharing between versions depends on user deletion behavior: if users delete all versions before each upload, for example, we will get no storage compression due to sharing between versions.

The 'both' column shows the average actual version storage compression ratio we achieved, including the savings from sharing within versions and across versions (of the same service instance). We achieved a storage compression ratio of 6.8% (4.0%) or 15x (25x). These numbers mean that 10 successive versions (one full and nine incrementals) can be stored by a JS in the space required to store one uncompressed version.

The astute reader will have noticed that our storage compression ratio is slightly worse than our upload compression ratio; this is because our URS upload code keeps a copy of the last (partial) upload in a staging area on the server; this reduces the amount of data that must be transferred, but does not count as previously stored data for the purpose of determining how much new data has been added to the store.

4.4 Speed

The median time from an animator requesting a version be uploaded to all of that version's bits being known to be present on the Jumbo Store (upload) is shown for each service instance in Figure 5 below; the average median time to upload a version (avg) was 4.4 minutes and the median time for all uploads (all) was 1.8 minutes.

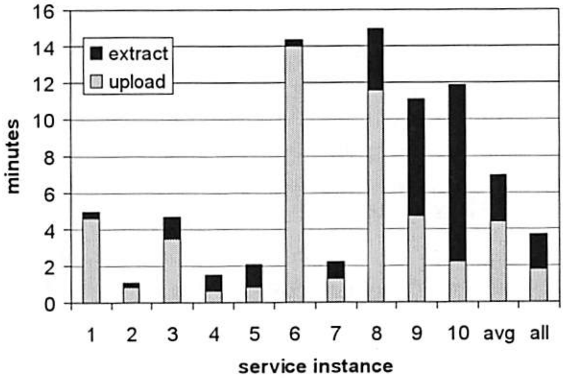


Figure 5: Median upload and extraction times

Also shown is the median time from the request until the new version is available in the URS's Version Cache (upload+extract), which is required before rendering can start. Extracting a version involves downloading that version from the Jumbo Store to the Version Cache located on the same machine. Because the Version Cache copy is uncompressed, extraction time is necessarily proportional to the uncompressed size of the version rather than the much smaller amount of data actually sent/stored in the Jumbo Store. Requiring extraction is suboptimal; in the future we may be able to eliminate it and the Version Cache altogether in favor of rendering directly from the Jumbo Store data via a filesystem abstraction. Extraction took 2.5 (2.0) minutes, yielding an overall transfer time of 6.9 (3.8) minutes.

To put this in perspective, downloading a single frame (~900 KB) took 10 seconds on average. Although an average of 250 frames were downloaded per version (~50 minutes of total download time), most of these would have been downloaded either in the background while working or overnight—a small sampling of frames usually suffices to find errors/verify changes. Rendering a frame took a few minutes to several hours depending on the complexity of the frame (e.g., fur slows things down). Frames can be rendered in parallel, however.

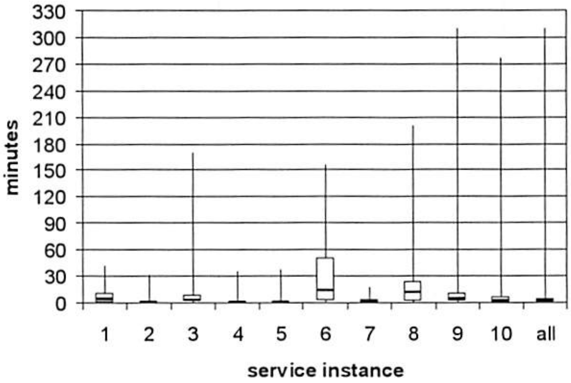


Figure 6: Distribution of upload

We quote median rather than mean values in this subsection because the underlying distributions are highly skewed toward smaller values; Figures 6 and 7 provide information about the distribution of upload for each service instance using box plots. Each box ranges from the 25th percentile value to the 75th percentile value and is divided into two parts by a line at the median (50th percentile) for value. Lines extend vertically from each box to the minimum and maximal values of the given distribution. The high tails of the upload distributions drop off roughly inversely to time.

Upload times are affected by the actual amount of bandwidth available and the amount of data that needs to be uploaded. Actual bandwidth, which we were unable to measure, depends on the speed of the animator's connection and the amount of congestion experienced from other programs on the same computer, neighbors in the case of shared connections (e.g., cable modems), and other users of the transatlantic cable. Except for two of the instances, most of the variance in upload times for an instance is due to variance in the amount of data that needed to be uploaded; Figure 8 shows the distribution of the sent user data size for each instance. The average median amount was 3.8 MB and the median amount for all uploads was 0.80 MB.

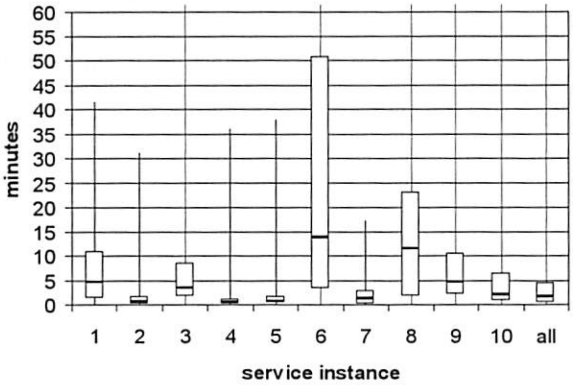


Figure 7: Detail of bottom of Figure 6

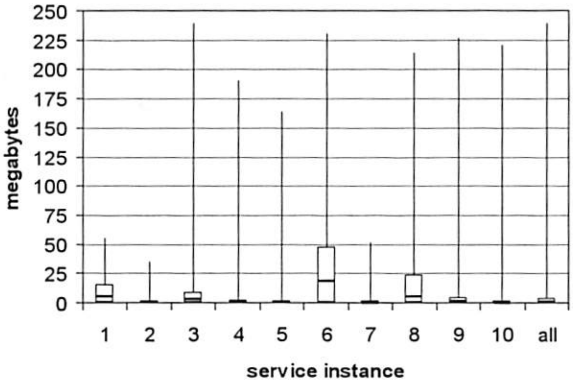


Figure 8: Distribution of amount of user data sent

Although we do not know what the actual raw maximum bandwidth available for any given upload was, we can estimate the effective bandwidth (total size of user data sent/time required) for each instance; Table 3 below shows the results of applying linear regression to each instance's sent user data size, upload time pairs excluding a few outlier points whose residual's were more than three standard deviations from the norm. For example, we predict service 1 sending 5 MB of file data would take 25 + 5*1024*8/187 = 244 seconds. Fit was good (high R^2) for all service instances except 5 and 7; recall that service 5 had numerous connection problems.

These bandwidth calculations do not include control messages, queries, metadata, or TCP overhead. Overhead includes both setup/finishing steps and work proportional to the size of the version being uploaded rather than the amount of data being transferred (e.g., queries).

service instance	bandwidth (Kbits/s)	overhead (s)	R^2
1	187	25	0.998
2	155	18	0.989
3	198	81	0.989
4	706	29	0.981
5	638	59	**0.307**
6	200	166	0.983
7	184	66	**0.601**
8	165	103	0.954
9	101	129	0.999
10	192	176	0.929

Table 3: Estimated effective bandwidth for each service instance

4.5 User feedback

Extensive interviews were conducted with the teams of animators before and after SE3D. We report here mostly the parts relevant to the use of the Jumbo Store in the URS. The interview subjects agreed unanimously that the URS was easy to setup and install; 33% thought it met expectations while 56% thought it was simpler and easier than expected. More telling, almost all subjects said they would be interested in it for commercial use. The faster rendering speed and the ability to be operated remotely of the URS led several of the animators to change their working practices; one animator was in The Hague for nearly 6 weeks and continued working by using his laptop in Internet cafés.

Animators are not technical people. They are very visual/tangible thinkers; this led to some difficulties with the programmer-influenced user model and interfaces. We discovered after SE3D was over that there was a fair amount of confusion on how uploads worked and what versions were. Some animators mistakenly thought upload time was proportional to the amount of data in their upload directory; this caused some of those to take care to "upload" only the fraction of the movie source relevant to a given rendering step by copying the relevant files from their actual source directory.

There was also confusion about the meaning of "version". In the mind of the animators, a version is a snapshot of a set of files defining a project that have reached some key milestone in the project. They were thus puzzled when a minor change produced a new version. The animators' normal work practice was to keep each revision of a given scene file by using related filenames (e.g., clouds.1, clouds.2, etc.); some insisted on this practice even though they thought (erroneously) that it was hurting their upload performance.

5 Comparison with Rsync

The best alternative to the Jumbo Store we know of for uploading files across a low bandwidth connection is Rsync [26], an open source utility that provides fast incremental file transfer. Accordingly, we compared uploading a subset of the SE3D data across the transatlantic cable using the Jumbo Store (independently, with no SSL) and using Rsync.

The data used was a subset of the versions uploaded by the animators; more precisely, the data is from a copy made during a maintenance window late in SE3D's life of the Jumbo Stores' data files. It is thus lacking any versions uploaded after or deleted before that point. Although this is the most representative data we have, it is likely less compressible than the actual sequence of versions uploaded during SE3D because it is missing intermediate versions. The data used contains 441 versions distributed as follows:

service instance	versions		service instance	versions
1	90		6	15
2	41		7	84
3	12		8	6
4	76		9	2
5	99		10	16

Note that we have very few versions for service instances 8 and 9.

The uploading was done from a 1.8 GHz Pentium 4 PC with 1 GB of RAM running Suse Linux 9.1 in Palo Alto, California to an 800 MHz Pentium III PC with 512 MB of RAM running Red Hat 9 Linux in Bristol, England. Both PCs are inside the HP corporate firewall, but the connection between them runs through the public Internet and over the transatlantic cable. Previous experiments indicate that the transatlantic cable is the bottleneck for this connection, with a peak bandwidth of slightly less than 2.7 Megabits per second (2800 Kb/s).

Our experimental procedure was as follows: for each service instance, we first emptied the destination directory (for Rsync) or store (for JS). We then uploaded each version belonging to that instance in turn in the order they were originally uploaded. Every upload for a given service instance other than its first thus had the potential to be an "incremental" upload. We used tcpdump and tcptrace to record the elapsed wall time and number of unique bytes sent (i.e., the total bytes of data sent excluding retransmitted bytes and any bytes sent doing window probing) and received of each upload. Due to time constraints (a full run through of all the data for a single method takes weeks), we were only able to repeat this procedure once per upload method.

Figure 9 below compares the number of unique bytes transmitted (i.e., sent or received) by Rsync, by our original Jumbo Store with the block retransmission bug fixed (JS), and by an improved version of the Jumbo Store (JS+), which we describe shortly. For ease of comparison, we present normalized numbers where Rsync's performance is designated as 1.0. In addition to per service instance numbers, we also show numbers for combining all the uploads (all, with emptying when switching instances) and the median of the instance numbers (med). Overall, JS transmitted 52% (med 53%) or 1/1.92 (med 1/1.89) of the bytes that Rsync did.

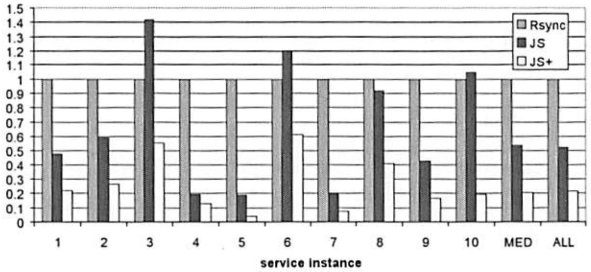


Figure 9: Total bytes transmitted for each method

We invoked Rsync with the "-compress" option, which is recommended for low bandwidth connections and has the effect of gzipping data before it is transmitted. This compression is on top of Rsync's delta compression, which attempts to send only the portions of files that differ. Our experiments indicate that failing to use -compress results in Rsync sending 190% more bytes overall (all) on this data set.

Inspired by this result, we created an improved version of Jumbo Store (JS+) that gzip's each set of chunks to be sent during transmission; by default, each set of sent chunks has 50 ~4 KB chunks for a total size of ~200 KB uncompressed. This change substantially increased performance: JS+ transmits 1/2.5 (med 1/2.6) the bytes that JS does and only 21% (med 21%) or 1/4.7 (med 1/4.8) of the bytes that Rsync did.

If we consider only the "full" uploads, JS+ transmits only 39% (med 55%) of the bytes that Rsync does. Considering only the "incremental" uploads instead, JS+ transmits only 15% (med 12%) or 1/6.7 (med 1/8.3) of the bytes that Rsync does.

Figure 10 below compares JS+'s performance to Rsync's using both bytes transmitted and time elapsed;

as with Figure 9, we have normalized so that Rsync's performance is 1.0. Measuring by time, JS+ is only 3.4x (med 3.1x) faster than Rsync. We estimate using linear regression that overall (all) actual bandwidth (unique bytes transmitted/elapsed time) was 2.44 Mb/s for JS+ and 2.49 Mb/s for Rsync with overheads of 6.8 seconds for JS+ and 3.0 seconds for Rsync.

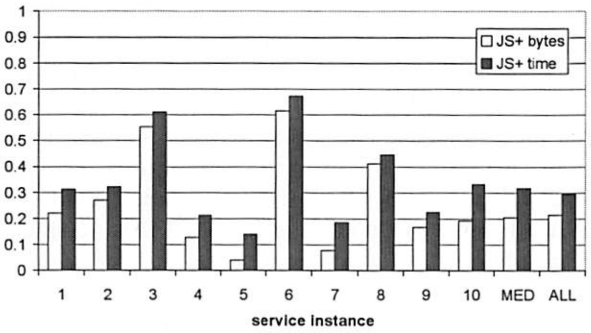


Figure 10: Normalized JS+ performance vs. Rsync

6 Discussion of results

The level of compression and reliability achieved by a system is heavily dependent on the actual data to be compressed and the setting it is deployed in: it is easy to achieve 100% reliability in a controlled lab setting or good compression by using synthetic data created from the same distribution your compression algorithm was designed to compress. SE3D represents the gold standard in test data: large amounts of real data collected over a long time from real users using the system for its intended purpose. Because the services were isolated from each other for security and performance reasons, SE3D can be viewed as a series of 10 natural experiments. The large variance in outcomes between experiments—the upload compression ratio varied by a factor of 38 and the storage compression ratio by a factor of 16, for example—indicates that animators differ greatly in the characteristics that affect our system and Rsync's performance. We expect our system to work as well or better for longer movies (the SE3D animators created ~5 minute shorts) because movies are built from short scenes and because there is more opportunity for reuse of characters, sets, and the like. The performance of Jumbo Store on other domains is currently unclear; we are conducting experiments to address this.

The reliability of the Jumbo Store itself once we fixed some initial bugs was perfect: all upload problems were due to the URS, either directly or indirectly (i.e., the need for restarts due to MOB limitations), or nonworking Internet connections beyond our control. Clearly, the animators could have benefited from a better explanation of how the upload process works: the error-prone process of managing separate upload and working directories used by some of them could have been avoided. Likewise, future versions of the URS should provide more workflow support and make a distinction between "major" (meaningful to animators) and "minor" (aka, JS) versions.

Aside from reliability, the most important metric for an upload system is average upload time. We estimate that our original system is 24 times faster than one that does no compression: without compression and at the observed effective bandwidths, the average service median upload would have taken 2.8 hours. The possible productivity improvements from switching from several hours per upload to several minutes should not be underestimated. Had we deployed instead our improved version of Jumbo Store (JS+), we estimate it would have speeded things up 1.5 times to 35 times faster than no compression and an average median upload time of 2.3 minutes (4.8 minutes with extraction). The variance in the amount of data that needs to be uploaded and hence the upload times is not too surprising if we consider the animation process similar to that of program development: the changes between program runs are mostly small, but occasionally the programmer makes a major change that cannot be tested incrementally.

The Jumbo Store—especially the improved version—clearly outperforms Rsync for the SE3D-derived benchmark. Primarily this is because JS+ sends only 1/5 the amount of data that Rsync does. We attribute much of this reduction to the JS's ability to exploit sharing across files with different names, both within versions and across versions. Because Rsync computes pair-wise delta's between files with the same path names, it cannot exploit this sharing. Although we did not investigate the causes of this sharing, it is clear that one cause is some animators' use of numbered file versions (e.g., "foo.1", "foo.2", etc.): because each new file version has a new name, Rsync sees no sharing.

When Rsync is used to upload data to Linux, hard links can be used to store multiple snapshots in a compressed manner [25]: if a file is unchanged from the last snapshot, Rsync can simply create a hard link to the last snapshot's copy instead of creating a new copy. This provides limited compression as even a one byte change prevents any sharing and there is no compression within files or between files with different names. The low degree of compression does mean that no extra extraction step would be needed if used with the URS.

7 Related work

Content-addressable stores (CASs) [5,10,11,15,19,24, 29] allow stored items to be retrieved by their hash. Flat CAS systems treat the items that they store as undifferentiated blobs: the interpretation of each item is entirely up to the store's clients. The Jumbo Store is a non-flat CAS system: while it does not interpret nodes' data fields, it is HDAG-aware and does interpret nodes' children pointers. This allows it to support important operations like 'compare by root' and version deletion that otherwise would require clients to perform thousands to millions of more basic operations, which is especially problematic over low bandwidth connections.

Venti [23], a versioned file store, and CFS [9], a read-only distributed file store, use HDAG-like structures at the application level but rely on a flat CAS for storing their data. SUNDR [19] and ROFS [15] use an HDAG encoding of directory structures to ensure the integrity of the contents on untrusted servers. They take advantage of the unique root hash property by signing just the root hash with the private key of a legitimate authority. Any client with access to the public key of that authority can then verify the integrity of the contents. An intruder without access to the authority's private key cannot modify the contents without being detected, since modifying the contents will change the root hash. These systems [9,15,19,23] do not use chunking or take advantage of the properties of HDAGs for facilitating directory synchronization. While SUNDR offers multiple versions, it does not seem to support the deletion of versions once a short time period has elapsed.

THEX (Tree Hash Exchange Format) [7] specifies a way to create a Merkle tree from a byte sequence, encode the resulting tree and encapsulate it in an XML file. Its main purpose is to allow verification of fragments of the byte sequence from different sources while trusting only one source to provide the root hash of the tree. It is meant to be used in conjunction with BitTorrent-like protocols to improve the detection and retransmission of corrupted blocks before the whole byte sequence is retrieved. Unlike our approach, THEX encodes the whole Merkle tree for a byte sequence in one message, so there is no sharing of intermediate nodes. As a result, compared to a flat representation of the block chunks, it actually increases the communication overhead for the file. THEX does not have any mechanism for encoding directory nodes.

Duchamp [12] describes a toolkit for synchronizing directory structures accessed as NFS mounts. A hash tree encoding of the structure of a directory tree, similar to our HDAGs, is used for facilitating the rapid synchronization of the 'master' and 'slave' directories. While Duchamp's toolkit supports the break up of large files into smaller pieces, it does not use chunking or indirect nodes for efficient file synchronization, and it does not support multiple versions. BitTorrent [4] uses fixed-sized blocks and compare by hash to transfer files.

Unlike these systems (Venti, CFS, SUNDR, ROFS, THEX, Duchamp, and BitTorrent), many recent systems including LBFS [21], CASPER [30], Pastiche [8], and TAPER [18] use chunking and compare by hash to optimize communication and/or storage requirements when multiple versions of a file exist. In the case of LBFS, this is done to speed up the transfer of files where the target may have already seen earlier versions of the files (or at least fragments of them). All of these systems use a flat sequence of hashes to represent a file and thus would benefit from the use of indirection nodes and HDAGs. They would also benefit from upgrading to our TTTD chunking algorithm.

TAPER [18] uses hash tree encodings of directory structures to facilitate directory synchronization. The hash trees used by TAPER are somewhat different from the HDAGs described in this paper. They do not encode the file and directory metadata, and as a result cannot directly be used for verifying the integrity of the directory structure on the target. The hash of intermediate directories is determined by an in-order traversal of all the children of the corresponding node, concatenating all the children's hashes as well as traversal direction information (e.g., H("up")), and taking the hash of the concatenation. This is a more computationally expensive procedure than that used by our encoding, with no apparent advantage. While TAPER uses chunking for file synchronization, it does not treat the resulting chunks as children of the file nodes in the hash tree. It uses a separate LBFS-like algorithm for file synchronization, and does not use indirect nodes to share sequences of long files. As a result, the whole hash sequence needs to be transmitted even if only one chunk has changed. TAPER does not support versioning.

Comparison with LBFS: Compared with LBFS, our combination of compare by root and indirection nodes significantly increases the bandwidth efficiency of transferring files. Where with LBFS the server has to be queried for every chunk, with our algorithm whole sub-trees of the directory structure can be skipped when an identical copy exists on the server. Moreover, because LBFS uses flat hash lists for its file representation, the whole file representation must be sent over the wire even if the modification to the file is small.

LBFS is a file-level protocol: it does not have any representation of the directory structure. As a result, directory data is neither compressed, nor verified, in its protocol. Our protocol, by contrast, which uses a HDAG-based representation of directory structure, is efficient, robust, and fault tolerant at the directory level. LBFS does not provide for the efficient storage of multiple versions of files or snapshots.

Note that distributed filesystems like LBFS are not suitable for the URS or many other synchronization applications because of their poor responsiveness (the trans-Atlantic cable has high latency), need for constant connectivity, and failure to respect the fact that the client's contents not the server's are the ground truth. Providing a disconnected mode would help but negates the primary value of using a distributed file system for synchronization: sending changes as they are made rather than all at once at the end. Supporting multiple operating systems is substantially more difficult with a distributed file system approach.

Comparison with Rsync: Even though Rsync is a directory tree synchronization protocol, it does the synchronization through pairwise file comparisons based on files' pathnames. As a result, it completely misses intra-source sharing (when multiple files in the source's directory tree share significant content) and is completely stumped when directories or files are renamed or moved. Our representation and algorithm are insensitive to such changes, and can naturally detect and exploit intra-source sharing when it exists. In terms of reliability and robustness, Rsync verifies data only at the sub-file level; it lacks any form of overall verification.

Comparison with Grid: Solutions exist in the Grid [14] community to synchronize, manage, and process data [1,2,6,14,28,31]. These approaches target a different problem: high-performance computing applications with relatively static, huge data sets (possibly terabytes), and (multi-)gigabit-class connectivity. Typical use cases in this environment do not require support for simultaneous, overlapped processing of multiple versions of frequently-updated input content.

8 Future Work

There a number of ways the Jumbo Store and URS can be improved:

Lazy extraction: Currently before a processing node can start rendering, the entire relevant version must be extracted from the Jumbo Store to the Version Cache. This can lead to significant delay as well as unnecessary work if not all of that version's files are needed for the current rendering task. A better solution would be to extract files only as needed directly from the Jumbo Store. Accordingly, we are working on a remote filesystem interface for JS so that clients (in this case the processing nodes) can directly mount read-only the filesystems contained in JS versions. It is not clear that this will entirely eliminate the cost of extraction as the lazy interface may be slower than directly accessing an uncompressed version due to poorer locality.

Trickle upload: The URS client currently sends changes only when the user explicitly requests an upload of a new version; consequently all the changes since the last upload must be transmitted before rendering can commence, leading to delays. A more responsive system would use trickle uploading where a background task periodically scans the user's data and optimistically sends any new data chunks to the Jumbo Store. When the user finally requests an upload, few chunks would likely remain to be sent, allowing rendering to start sooner. Sent chunks that were superseded by later changes would be freed later during garbage collection.

Larger multi-user stores: Our current Jumbo Store server uses an in-memory chunk index, which limits its holding capacity to tens of gigabytes (compressed) assuming ~4 KB chunks. While more than adequate for a single SE3D service, other utility computing services may have larger jobs or wish to share a single JS instance between many services. To handle this, we are developing a new JS server that uses a disk based index and has support for access control and allocating resources among users.

9 Conclusion

In this paper we described an HDAG-aware content addressable store, the Jumbo Store. An HDAG is an immutable data structure for representing hierarchical data where hash pointers are used to connect the nodes. We built an incremental upload mechanism for directory snapshots that takes advantage of the unique root hash, automatic self assembly, and automatic structure sharing properties of HDAGs and the store's HDAG support, to efficiently and reliably upload large directory snapshots over slow and unreliable public internet connections. The store has built in facilities for the creation, retrieval and deletion of versions, which are named HDAGs. We used these facilities to build a system for efficiently storing many versions of a directory tree.

The ability to transmit large quantities of data over the slow Internet connections typical of many organizations, to be processed by Utility Services, is often perceived as a barrier for widespread adoption of the utility model. The JS was successfully used within a Utility Rendering Service, used to create 3D animated movies, and demonstrated that interactive, data-intensive services can work well even over low-bandwidth connections. The speed of upload offered by the storage system encouraged users of the service to work in an experimental fashion to try new ideas containing variations of data content. The synchronization and storage performance of the JS with the real-world data produced by small teams of animators has been analyzed and compares favorably with other competing approaches, both in the URS environment and under controlled experimental conditions.

References

[1] W. Allcock et al. Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing. In *Proceedings of 2001 IEEE Mass Storage Conference, 2001.*

[2] W. Allcock, J. Bester, J. Bresnahan, S. Meder, P. Plaszczak, and S. Tuecke. GridFTP: Protocol extensions to FTP for the grid. GWD-R (Recommendation), April 2002. Revised: Apr 2003, http://www-isd.fnal.gov/gridftp-wg/draft/GridFTPRev3.htm.

[3] Autodesk Maya. http://www.autodesk.com/alias. Maya is a registered trademark of Autodesk, Inc.

[4] BitTorrent: http://www.bittorrent.org/protocol.html

[5] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pp. 13-24. Seattle, WA (August 2000).

[6] D. Bosio, et al. Next-Generation EU DataGrid Data Management Services. *Computing in High Energy Physics (CHEP 2003)*, La Jolla, California, March 24–28, 2003.

[7] J. Chapweske. Tree Hash Exchange Format (THEX) http://www.open-content.net/specs/draft-jchapweske-thex-02.html

[8] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of OSDI: Symposium on Operating Systems Design and Implementation* (2002).

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01).* Banff, Canada, Oct 2001.

[10] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS 2002)* (July 2002).

[11] P. Druschel and A. Rowstron. A. PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of HotOS VIII*, pp. 75–80.

[12] D. Duchamp. A Toolkit Approach to Partially Connected Operation. In *Proc. of the USENIX Winter Conference*, pp. 305-318, Anaheim, California, Jan. 1997.

[13] K. Eshghi and H. K. Tang. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. HP Labs Technical Report HPL-2005-30R1, http://www.hpl.hp.com/techreports/2005/HPL-2005-30R1.html

[14] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid Computing: Making the Global Infrastructure a Reality. *The Physiology of the Grid*, Wiley, 2003, pp. 217–249.

[15] K. Fu, M. Frans Kaashoek, and D. Mazières. Fast and secure distributed read-only file system.
In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pp. 181-196, Oct 2000.

[16] Val Henson. An Analysis of Compare-by-hash. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems* (HotOS IX), Lihue, Hawaii, May 2003, pp. 13-18.

[17] HP Utility Rendering Service: http://www.hpl.hp.com/SE3D/whitepaper-urs.pdf.

[18] N. Jain, M. Dahlin, and R. Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *Proc. of the 4th Usenix Conference on File and Storage Technologies (FAST)*, Dec 2005.

[19] Jinyuan Li, Maxwell Krohn, David Mazieres, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, pp. 91–106.

[20] R. Merkle. *Secrecy, authentication, and public key systems*, Ph.D. dissertation, Dept. of Electrical Engineering, Stanford Univ., 1979.

[21] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-Bandwidth Network File System. In *Proc. of the 18th ACM Symposium on Operating Systems Principles.* Chateau Lake Louise, Banff, Canada (October 2001).

[22] C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *Proceedings of the General Track, 2004 USENIX Annual Technical Conference.*

[23] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies* (2002).

[24] A. Rowstron and P. Drushel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware).* Heidelberg, Germany (November 2001).

[25] M. Rubel. Easy Automated Snapshot-Style Backups with Rsync.
http://www.mikerubel.org/computers/rsync_snapshots/

[26] Rsync: http://samba.anu.edu.au/rsync/

[27] SE3D: http://www.hpl.hp.com/se3d

[28] H. Stockinger et al. File and Object Replication in Data Grids. In *Proceedings of 10th IEEE Intl. Symp. on High Performance Distributed Computing.* 2001.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *Proceedings of the ACM SIGCOMM 2001.* San Diego, CA (August 2001).

[30] Niraj Tolia, Michael Kozuch et al. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proc. of the General Track, USENIX 2003 Annual Technical Conference*, pp. 127-140.

[31] W. Watson III, Y. Chen, J. Chen, and W. Akers. Storage Manager and File Transfer Web Services, *Grid Computing–Making the Global Infrastructure a Reality.* Wiley, pp. 789-801.

Data ONTAP GX: A Scalable Storage Cluster

Michael Eisler, Peter Corbett, Michael Kazar, and Daniel S. Nydick
Network Appliance, Inc.

J. Christopher Wagner
IronPort Systems, Inc.

Abstract

Data ONTAP GX is a clustered Network Attached File server composed of a number of cooperating filers. Each filer manages its own local file system, which consists of a number of disconnected flexible volumes. A separate namespace infrastructure runs within the cluster, which connects the volumes into one or more namespaces by means of internal junctions. The cluster collectively exposes a potentially large number of separate virtual servers, each with its own independent namespace, security and administrative domain. The cluster implements a protocol routing and translation layer which translates requests in all incoming file protocols into a single unified internal file access protocol called SpinNP. The translated requests are then forwarded to the correct filer within the cluster for servicing by the local file system instance. This provides data location transparency, which is used to support transparent data migration, load balancing, mirroring for load sharing and data protection, and fault tolerance. The cluster itself greatly simplifies the administration of a large number of filers by consolidating them into a single system image. Results from benchmarks (over one million file operations per second on a 24 node cluster) and customer experience demonstrate linear scaling.

1 Introduction

File storage is divided between local file systems and network file systems. As networks have become faster and more reliable, network file systems have become an important aspect of most organizations IT infrastructure. Typical applications include home directories, databases, email, and scientific and technical computing.

The widespread deployment of network file systems has led to the development of specialized file server solutions, commonly referred to as Network Attached Storage (NAS). NAS systems, generically called **filers**, have typically been monolithic systems, with a single or dual controller or head, fronting a large amount of disk. Such systems often support virtualization, allowing the aggregated disk storage to be divided into a number of virtual volumes, allowing the virtual volumes to be presented through a number of virtual servers, all hosted by the same filer hardware.

The limits of this approach are obvious. As the number of client machines attached to networks increases, the number of filers must increase commensurately, or the filers will become overloaded. A filer is based on similar hardware to a client, and so, the only way for a filer to "keep up is to either add more filers, or to increase the performance of the filer. The second option is expensive; specialized hardware solutions seldom maintain a performance advantage over commodity hardware.

But adding more filers has the disadvantages of being complex to administer, disallowing opportunity for load balancing and sharing among the filers, and requiring the clients to mount a large number of different filer volumes.

We decided that the best solution to this problem is to cluster a number of individual filers to form a single file server. For this purpose, we developed GX, which leverages the existing ONTAP-7G architecture [NetApp], but adds a switched virtualization layer just below the client-facing interfaces. This allows the storage of a large number of filers to be presented as a single shared storage pool. The key features provided are scalability, through the ability to add filers to the cluster, location transparency of data within the cluster, an extended namespace that can span multiple filers, increased resiliency in the face of failures, and simplified load and capacity balancing.

2 Related Work

GX draws on much previous work. It uses a remote file system switching mechanism inspired by the Virtual File System (VFS) [Kleiman]. GX supports both NFS

[Sun] and CIFS [SNIA]. GX provides many benefits which are drawn from the Andrew File System [Howard], and its commercial successors AFS [Campbell] and DFS [Kazar1990]. The AFS namespace connected multiple disparate AFS servers, each of which stored a **cell**. Each cell could store multiple volumes, and volumes were linked to each other through internal mount points, with the linkage and location of the volumes defined by the Volume Location Database. AFS also provided benefits such as location transparency, and the ability to load balance beneath a single client mount point.

However, AFS and DFS required special client code. The experience of AFS and DFS in trying to establish a client footprint was one of the main observations that motivated the use of NFS and CIFS as the client access protocols in GX. The goal of GX was to provide the benefits of AFS and DFS, while providing client access through the widely deployed NFS and CIFS protocols.

The GX architecture is inspired by that of Spinnaker Networks [Kazar2002]. Spinnaker was acquired by Network Appliance in 2004.

Frangipani is a SAN-type file system based on a distributed lock manager coordinating accesses from a collection of file system clients to a shared virtual disk [Thekkath]. We decided against building a SAN file system because of concerns about the overhead of distributed lock management in workloads with read/write data sharing, or high volumes of meta-data updates of any kind. We were also concerned about the size of the failure domain in such an architecture, where a bad piece of hardware could cause almost unbounded damage.

Slice has a goal to distribute the directory operations across many servers without partitioning the namespace into volumes, which avoids user visible mount points, re-partitioning volumes if volume loads grow at uneven rates, and avoids the issue of hard link and rename crossing volume boundaries [Anderson]. Slice distributes every object separately, based on a hash of its file name. Thus name lookups distribute well across the cluster. Hard link creation, rename, and file removal require a two phase commit across servers. We ruled out such an approach due to the overhead of distributed transactions.

3 Architecture

The architecture of GX addresses a number of key challenges in clustered NAS service.

First, we wanted GX to provide horizontal scaling, so that a cluster could grow over time to provide additional storage and processing capabilities for the namespace exported by GX.

Second, we wanted location independence, so that data could be reconfigured dynamically while the system was operating.

Third, we wanted to abstract the externally visible notion of a "server away from the physical hardware. This would allow us to provide multiple virtual servers within the clustered system, so that the actual set of physical resources dedicated to a particular name space can be chosen to match what is required for this name space, and does not have be reserved in any special fixed size units, such as entire disks or network interfaces.

Fourth, we wanted to pay as little as possible for these features.

3.1 Overview

The GX architecture is a high level switched architecture, where network file system requests are received by a file server s front end, mapped into one or more simple file system requests, and then transferred over a cluster fabric to the server that stores the data.

Data is stored in **volumes**, which are file system sub-trees consisting of an inode file with a root inode and a set of directories and files contained under that root inode. An **aggregate** is a collection of volumes, which can be thought of as a virtualized UNIX disk partition.

A **namespace** is composed of volumes, joined together by **junctions** which are entries in a volume that act as mount points for other volumes. Section 4 discusses namespaces and junctions in more detail.

Figure 1 shows three aggregates, containing a number of volumes, spliced together to make a single namespace.

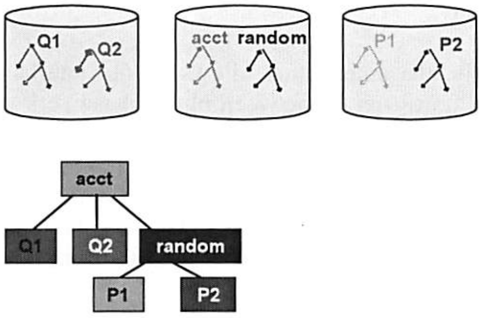


Figure 1: Example namespace

Junctions in the root volume for this namespace (*acct*) lead to volumes *Q1*, *Q2* and *random*, and junctions in random lead to volumes *P1* and *P2*.

File servers in this model, are divided into three components. Requests are received at virtual interfaces (or **VIFs**), each with its own IP address and network routing domain. These requests are initially processed by the **N,** or **networking blade**, which terminates incoming NFS and CIFS connections and maintains protocol specific state (such as CIFS connection state). The N-blade translates the incoming requests into SpinNP remote procedure calls, which are transmitted over a cluster fabric to the server responsible for the target volume. These SpinNP file system calls are in turn processed by the **D,** or **data, blade** on the target server. SpinNP file system requests can be thought of as RPC-based versions of a Vnode layer, augmented to handle the locking complexities encountered when simultaneously supporting the Microsoft CIFS protocol, NFSv3, NFSv4 and iSCSI, among other protocols. Figure 2 shows the internal structure of a two element cluster.

There are two slowly changing cluster-wide databases used to route requests and responses to the appropriate modules in the cluster. First the **volume location database (VLDB)** tracks both the identity of each volume s containing aggregate, as well as the D-blade that is currently responsible for that aggregate. The N-blade consults the VLDB to determine which D-blade to send a request for a particular volume. In addition, the D-blade occasionally needs to initiate callbacks to a client via a particular virtual interface. Second, the **VIF manager database** tracks which N-blade is currently hosting each virtual interface. In today s protocols, these callbacks are typically issued in support of such operations such as CIFS oplock revokes, NFSv4 delegation revokes and NLM asynchronous lock grants.

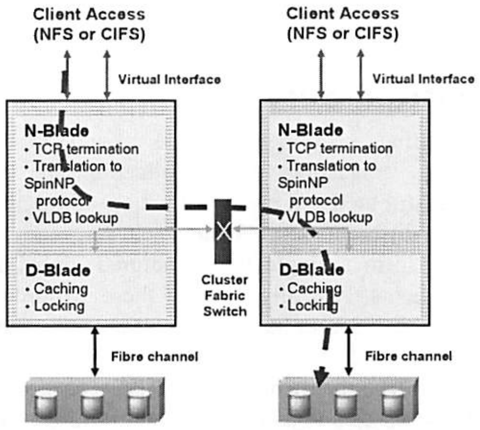


Figure 2: Example Cluster

Figure 3 shows three aggregates, one behind each GX server. Each aggregate stores two volumes which contain the junctions to produce the tree structured namespace shown on the upper right.

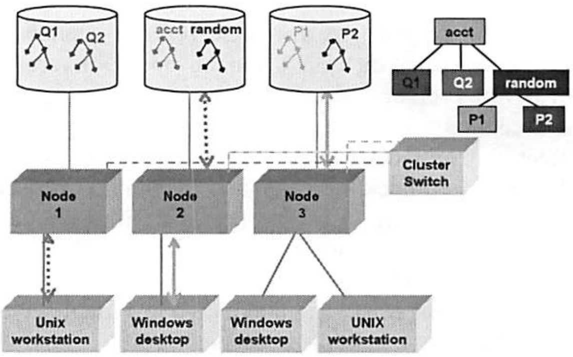


Figure 3: Overall architecture

We now walk through the details of the processing of an operation. Assume a storage client sends a CIFS request to filer node 2 with a file handle specifying a file in volume P1. The N-blade on server node 2 removes the request from the queue of the CIFS connection, and extracts the SpinNP file handle from the CIFS state and the request. The N-blade extracts a volume ID from the file handle and uses this ID to index into a cached copy of the VLDB to find the ID of the aggregate storing volume P1. Then the N-blade uses the aggregate ID to lookup the network address of the D-blade responsible for aggregate (node 3) and sends one or more SpinNP requests to that addresses. The D-blade receives the SpinNP request and executes it, sending the response back to the originating N-blade, which generates a CIFS response and sends the response back to the client. Section 5 discusses SpinNP in more detail.

3.2 Virtual Servers

The preceding section described the GX data architecture, consisting of a collection of virtual volumes distributed among a collection of aggregates owned by different servers in a cluster, and glued together to form a single tree-structured global name space. Clients typically access these volumes by contacting the server at one of several network addresses (typically IP addresses). A **virtual interface** (VIF) is a virtual network card having one or more network addresses and a corresponding routing domain, bound at any instant to a physical network card. Virtual interfaces migrate to different physical network cards in case of failures to links or servers.

This entire collection of virtual volumes accessed via a set of virtual interfaces may be virtualized as **a virtual server**. A virtual server consists of its own set of virtual volumes, with one designated as the virtual server root, acting as the root of the virtual server s private name space. A virtual server also contains its own set of virtual interfaces, and any operation received on a virtual server s interface is automatically restricted to accessing one of the virtual server s volumes. In effect, this provides GX with the capability to divide a cluster s resources into isolated sections, with each section on its own private subnet, and with users from that subnet limited to accessing data in their own section, independent of any discretionary access control lists that might exist. The alternative to virtual servers would be to have multiple clusters.

3.3 Location Independence and Single System Image

A fundamental goal of the GX architecture is the aggregation of a set of servers into a cluster that appears externally as a single large server having many volumes and many network interfaces through which those volumes may be accessed. A fundamental property of these clusters is that any volume can be accessed via any interface in the cluster, and any element can be managed via any network interface in the cluster (a well-defined exception is described in section 3.2).

Location independence provides the core underpinnings for transparent and online resource reconfiguration. Specifically, volumes can be moved dynamically between aggregates and servers in a GX cluster, and these move operations occur completely transparently to the clients, moving both data and file lock state atomically.

Similarly, the architecture allows virtual interfaces to migrate transparently between physical network cards, although today s implementation of VIF migration is not completely transparent for certain protocols: CIFS users see a TCP connection reset that may be visible as a very short lived server failure. This may be remedied by migrating the CIFS TCP connection state before doing, at least, a manual VIF failover.

The result is a clustered file system where data can be transparently moved between nodes in a cluster, users can be transparently moved between nodes in a cluster, and in general, the entire system can be reconfigured online. This level of management flexibility is required in today s environments where no suspensions of access to storage are permitted, even for adding new servers or decommissioning obsolete servers.

4 Namespace

4.1 Overview

The namespace uses ideas from the Andrew Filesystem and AFS for constructing a namespace built from a collection of storage volumes linked to each other in a tree. AFS and the GX namespace share the following properties:

- A consistent view of the namespace is provided from any client.
- Different volumes of the namespace tree can come from different server nodes on the network (termed D-blades).
- Volumes can be moved among nodes without disrupting clients or processes on the clients holding open files of the migrated volume.
- Volumes can be replicated and replicas distributed across the server nodes for purposes of load balancing and enhanced data availability.

AFS and GX diverge in that the latter does not require special purpose client software to access the namespace or to enjoy the properties described above. Instead existing file access mechanisms like CIFS and NFS can be used.

While AFS maintained its namespace via pointers to child volumes stored in the filesystem, GX uses a

junction table (maintained in the VLDB). The junction table contains **mounting relationships,** which are triples consisting of a parent volume, a child volume, and a **junction**, which is a reference to a directory-like file object that exists in the parent volume. The junction, identified by its inode and generation numbers, serves as the mount point for a child volume. A volume, identified by a **Master Data Set Identifier (MSID)**, can have multiple junctions in it, each corresponding to a mount point in the junction table. A child volume can be a parent volume to other child volumes. The junction table thus can be thought of as table of directed arcs in a graph.

Some limitations of volume mounting include:

- Only the root directory of a child volume is mounted on a parent volume's directory.
- A child volume cannot have multiple parent volumes (i.e. there may be no more than one entry in the junction table specifying this volume as a child).

The former limitation exists partly out of expediency and partly from the observation that volumes can be very small in ONTAP GX (as is the case in ONTAP-7G) [NetApp]. There's no need to mount subdirectories of a volume, when one could instead break a volume into several smaller volumes (a procedure which is simplified via the use of zero-copy volume cloning). In addition, the root of a volume is the obvious place to look for a parent volume to which to ascend.

The latter limitation exists to support traversal between a child volume and its parent volume as happens with a UNIX "cd .. issued from the root directory of a child volume. Otherwise it is not clear which parent volume to ascend to.

When an N-blade gets a request to access an object name that is a junction, it checks to see if there is a child volume mounted on the junction, instead of returning an NFS file handle that contains the MSID of the volume the junction lives on, and the fileid and generation number of the junction. The N-blade queries the junction table, using the MSID of the volume (the parent), and the inode number and generation number of the junction. If an entry is found, (and there should be at most one entry), then the entry has the MSID of the child volume, and the VLDB is queried with the MSID as a key to find the D-blade in the GX cluster that owns the child MSID.

When a D-blade gets a request to return the inode number and generation number of the parent of the root of a volume, it indicates to the N-blade that the root of the volume has already been reached. This time the N-blade needs to query the junction table using the child volume's MSID as the key. The result of the query is either the MSID of the parent volume, and the inode number and generation number of the junction the child volume is mounted on, or the result is an indication that the client is already at the root directory, of the root volume, of the namespace tree.

In the event of a D-blade failure, the ability to transit to any volume on that D-blade is lost.

The goal of the namespace is to emulate a single filesystem. However in some areas the emulation breaks down. GX does not support renaming or hard linking files across volumes. Other areas of break down are described in the next two subsections.

4.2 Considerations for CIFS Clients

CIFS supports a form of access control called Access Control Lists (ACLs). Each file can have one ACL. Each ACL contains one or more Access Control Entries (ACEs). An ACE contains a user or group identifier, and a bit mask of operations (read the file, delete the file, write the file, etc.), specifying whether a user or member of the group is to be allowed to perform the operation or to be denied the operation, and some flags. One flag is the inheritance bit. If this bit is set on a directory, then this indicates that any file or directory created inside the directory should inherit the ACE.

An issue is whether ACL inheritance should work across junctions. We considered and rejected two possibilities.

The first possibility is if the junction's parent directory has the inheritance bit set, then the ACL should be inherited in the child volume, even if the root directory of the child volume does not have inheritance set.

The second possibility is if the root directory of the child volume has the inheritance bit set (even if the parent directory of the junction of the parent volume does not have inheritance set) the ACLs should be propagated from the parent volume. The ability to dynamically reconfigure the namespace complicates these possibilities – what do we do if a volume is remounted under a directory with a different inheritance value?

In the interest of producing the least surprise, we have chosen to not inherit ACLs across junctions, regardless whether the inheritance bit is set in the parent volume or the root of the child volume.

CIFS supports a feature called "Change Notify" where a CIFS client can register an interest in being notified about changes to the directory or an entire tree of directories and files. For applications it arguably makes sense for change notify to work across junctions. However, because this has the potential to consume almost all cluster bandwidth, GX does not support change notify across junctions.

Both ONTAP-7G and GX support a feature called the security style which applies to volumes. The security style indicates whether to use Windows (ACLs), UNIX (mode bits), or a mix of the two for enforcing access. As in ONTAP-7G, the security style can be assigned per volume in GX . However if the security style is not specified when a volume is mounted, it will inherit the security style of its parent volume.

4.3 Considerations for NFS Clients

NFS clients typically run on systems that expect some conformance to UNIX semantics. One semantic used by UNIX utilities like "cp", and "mv" is to invoke the *stat()* system call on two or more files and use the returned inode number to ensure the files are different. Among other data, the *stat()* system call returns the inode number of the specified file. Suppose "m" is mount point, and "p" is a file in the parent volume, and "c" is a file in the child volume. Because the child and parent volumes are independent volumes, "p" and "c" could very well have the same inode numbers. In that case, "cp p m/c" would fail because the inode numbers of the source and target are the same. We considered and rejected using the full 64 bits of an NFS version 3 inode number to encode the MSID and inode number, because not all clients, or applications on such clients can cope with the upper 32 (or 33) bits of an inode number being non-zero. Instead, we borrow the AFS idea of computing a hashed inode number using the file's volume's MSID and the file's inode number as input. This computed inode number is used only for replies to GETATTR (get attribute) requests; the NFS file handle uses the real inode number.

We found that some NFS clients had problems when the link count of a directory did not exactly equal the number of child directories plus the entries for "." and "..". This was caused by a junction in a directory not increasing the link count. We now implement junctions as objects that behave like directories in all ways, except that permissions for search, read, and write are denied to all. This way, directory properties like increasing the link count of its parent directory are preserved.

4.4 Snapshots

WAFL snapshot semantics continue to be supported in GX with some ramifications for the global namespace [Hitz]. Supporting coordinated snapshots among volumes distributed among several D-blades would be desirable, and theoretically possible (if difficult). Snapshot coordination is currently not part of GX architecture because coordinating snapshots across all volumes of an arbitrarily-sized namespace cannot scale.

The magic ".snapshot directory is omnipresent in GX, but because snapshots are not coordinated, the system denies clients the capability to navigate from a snapshot of a parent directory into a snapshot of a child volume. The reason is illustrated by an example. At time T a snapshot of parent volume A is taken, at T+1, child volume B is unmounted from A, and C is mounted where B was, at T+2 a snapshot of C is taken, and at T+3, the user unexpectedly encounters a snapshot of volume C instead of a snapshot of B.

5 SpinNP

The GX cluster is connected by a family of message-passing protocols called **SpinNP**. SpinNP is a layered system that defines a session and operations layer, and that specifies the requirements of its underlying transport layer. SpinNP is used for all high-traffic message passing within the cluster, both between N-blades and D-blades, as well as between different D-blades.

In developing SpinNP, we were motivated by the need for a RPC protocol with a tightly integrated session layer. We took a holistic approach to the design of SpinNP, which gave us the ability to assign function to the transport, session, and application layers as needed. We were able to build security, flow control and versioning features in from the ground up, and to design a protocol that can be run over multiple different and commonly available transports.

5.1 Transport Requirements

SpinNP sessions are layered over a network transport. Any transport can be used as long as it provides a connection-oriented protocol that provides reliable delivery of completely framed messages. We built a transport called RC, which is a simple message framing layer over UDP. The framing is required to delimit SpinNP messages on a byte stream protocol such as TCP. We also implemented other transports, including a memory-to-memory transport which is used only in the case of local communication between two blades which are co-located on the same cluster node. It is also possible to implement a transport over the PCI or other I/O buses.

Each transport provides an abstraction of one or more connections. Connections are simply pipes through which we can feed messages. Connections supply resource-limited flow control at the level of individual messages. More than one connection can be in operation simultaneously between the same pair of communicating entities.

The transport must be robust enough to inform the session layer when a connection has been lost. Timely notification of connection loss is important to speed session recovery.

5.2 Sessions

SpinNP sessions provide: strong security based on the GSS_API [Linn]; outer level flow control; the capability of exactly-once message delivery; strong major and minor versioning support; and multiple message priority channels per session.

SpinNP operations are RPC-like, and consist of one request message and a corresponding response message that is returned from the receiver to the sender of the request. SpinNP sessions support a bi-directional request flow and corresponding response flows in the opposite directions. Each SpinNP session is layered over at least three connections, one each for message priority levels low, medium and high. Low priority is used for normal request and response traffic. Medium priority is used for callback requests and responses. High priority is used for normal requests that are performed as a result of a callback. An example is the data flushes that occur after a file delegation is revoked by a callback.

The session is initially formed by the exchange of a SpinNP _CREATE_SESSION request and response. This request and response have distinguished values in their first bytes, which allow negotiation of the overall SpinNP protocol level. Once a mutually agreed-upon base protocol version has been established, the connection is authenticated by the exchange of GSS tokens. This follows the normal GSS sequence of security context creation (GSS_Init_Sec_Context), data protection and/or encryption (GSS_Wrap), and context destruction (GSS_Delete_Sec_Context). SpinNP has session level operations for each of these GSS commands. Session initialization also requires the negotiation of a set of operational protocols, called **interfaces**, which actually perform the operations that implement scalable file services within a cluster. Each interface consists of a set of operations, each defined by a request and response message pair. One of the primary interfaces is the file operations interface, which is used primarily between the N-blades and D-blades to implement network file system operations.

Each interface is separately versioned, independently of the base protocol. The base protocol version is negotiated during the exchange of the first two request and response messages. If agreement is reached on a mutually acceptable version, the session negotiation can be completed. This establishes the content of the message headers, as well as the content of a special set of requests and responses, called the session operations interface. These operations are used to perform GSS_API session initiation, and other SpinNP session negotiations.

The GSS_API is used not only to authenticate each participant in the session to the other over the initial connection, but also to authenticate each additional connection that is bound to the session. This is accomplished by the exchange of a pair of challenge-response tokens over the newly established connection. This exchange must complete before the connection can be bound to the session.

SpinNP sessions implement request level flow control. This limits the number of requests that can be outstanding in any session. The flow control combines slotted and sliding window techniques. Each session consists of a number of channels. Each channel has a sliding request window. Within a channel, requests are assigned sequence numbers. If the window size is *n*, the request with sequence number *i* +*n* cannot be issued until the response has been received for all requests with sequence number less than or equal to *i*. This ensures that a number of requests can be in-flight at the same time, allowing a high degree of concurrency. However, the sliding window is vulnerable to a single long-running request holding up the entire channel. To reduce the likelihood of this happening, SpinNP allows the creation of multiple channels for each session. Each channel has its own sliding window, and so if one channel becomes blocked, requests can still be sent over another channel. The end-to-end windowing allows the receiver to apply back-pressure on the request channels.

There are three priority levels of channels in a session: high, medium and low. There is no direct binding of channels to particular connections, except that the following rules are applied to select a connection on which to send a message. Low priority requests and responses are restricted to the low priority channels and are sent over low priority links. Medium priority messages are sent over the medium priority channel, and can be sent over either the medium or low priority connections. Similarly, high priority messages are sent over the high priority channel, but can then be transmitted over any of the connections, high, medium or low priority.

Sessions implement an at-most-once semantic, and will not tolerate lost messages from the transport layer. Should the transport layer fail to deliver a message, the session layer can trigger session recovery, which will attempt to construct a new session to replace the failed session. It is expected that the transport will retry sending the message until it determines that it is not possible to send the message. At this point, the transport will inform the session layer that a message is undeliverable, and the session will be reset.

5.3 File Operations

The most important SpinNP interface is the File Operations interface. The requests in this interface support the CIFS and NFS network file operations. Incoming CIFS and NFS messages are translated by the N-blade to SpinNP messages, which are in a common format independent of the original source format. SpinNP file operations support all the normal CIFS and NFS file access operations, as well as locking operations.

SpinNP file operations are designed to encompass the semantics of both CIFS and NFS, including NFS versions 2, 3 and 4. It does this without relying on an indicator in the request of what the source protocol was. (Such an indicator is included in each request, but is only used for statistics gathering, quality of service monitoring, and failure diagnosis.) This requires that all expected responder behaviors be specified directly in the protocol. For example, the protocol requires that file names be accompanied by a set of flags that indicate whether case sensitive or case insensitive naming applies. Much of the challenge of implementing a server is to ensure correct operation under arbitrary source protocols without relying on the identification of the specific source protocol. An additional challenge is to select the correct behavior when the semantics of two clients, using different network file service protocols, conflict. Such conflicts are usually resolved consistently, by conventions favoring one or the other source protocol.

The file operations support all NFS and CIFS semantics, including CIFS oplocks and NFSv4 delegations. To support these features and others, a set of file callback operations is also defined. File callbacks flow from the D-blade to the N-blade, which relays them to a client, which in turn responds to the N-blade, which in turn responds to the D-blade. These callback operations are used to set the client state, for example, to recall a file or directory delegation. Such operations are necessary to ensure that the server maintains control over its resources and to give the server the ability to inform the client when it is expected to synch cached updates.

The full set of SpinNP file operations resembles NFS, and is listed below. With few exceptions (e.g. WATCH is for setting up change notify, and is specific to CIFS) NFS and CIFS requests can be handled with the same operation. The specifics of NFS or CIFS semantics are specified via flags and parameters in each request. E.g. WRITE includes a flag to support CIFS called RSRV_AT_EOF which directs the D-blade to reserve extra disk space for the file whenever a write is done that extends the size of the file.

Table 1 SpinNP File Operations

NULL	READDIR
ACCESS	READLINK
CLOSE	REMOVE
CREATE	RENAME
DISCARD_CRED	SCAN_MATCHING_LOCKS
DOWNGRADE	SCAN_FILE
FH_TO_PATH	SETATTR
FOLD_FILE	SET_CRED
GETATTR	SHARE
GETATTR_BULK	UNLINK
GET_CRED	UNSHARE
GET_PARENT	WATCH
GET_ROOTFH	WRITE
LINK	
LOCK	**call back operations**
LOCK_GRANTED_ACK	NULL
RECLAIM_LOCKS_ACK	LOCK
LOOKUP	MOVE_LOCKS
MOVE_LOCKS	NOTIFY
OPEN	RECALL_LOCK
PREFETCH	RECLAIM_LOCK
READ	RETRY

While we originally planned to build a replay cache into the SpinNP session layer, we discovered we could achieve better end-to-end resiliency by building a replay cache in the SpinNP file operations layer, and that this can also provide the functionality required in an NFS response cache. It proved to be simpler and more effective to provide a single end-to-end solution than to chain together multiple links protected by different replay caches.

5.4 SpinNP IDL

We created a new interface description language (IDL) for SpinNP. This IDL has features that facilitate direct compilation of marshalling and unmarshalling code from the SpinNP specifications. The IDL relies heavily on two variable structures. The first is the **switch**, which selects variable fields from one of several alternatives. The switch is comparable to a C union; however, the selection of a branch is explicitly determined by a discriminator which is the first element of the switch construct. The total size required by a switch is determined by the selected branch, not by the size of the largest possible branch. The other is the **collection**, which can select any number of variable fields, specifying which fields are selected by a bitmap specification which appears at the beginning of the collection.

Examples of a switch and a collection from the SpinNP file operations specification are:

```
struct fileop_req_msg_body_t {
  src_protocol_t              src_protocol;
  fileop_request_flags_t      fileop_flags;
  cred_t                      cred;
  switch (fileop_proc_num_t proc) {
     case NULL_FILEOP:
       null_fileop_req_t      null_fileop;
     case ACCESS:
       access_req_t   access;
     ...
  } s;
};

collection file_attr_t
     (file_attr_types_t fa_type) {
  case FA_CHANGE:
    uint64_t            fa_change;
  case FA_SIZE:
    uint64_t            fa_size;
  ...
};
```

The SpinNP IDL is strongly typed, which facilitates compilation of code directly from the specification. We developed an IDL compiler to build C, C++, Perl and Ethereal code all from the same input file, which is the written specification of the interface. Having an exact match between the protocol specification and the code, including Ethereal descriptions, significantly sped development [Lamping].

One of the most important features of the SpinNP IDL is its strong support for versioning. It is possible to annotate a single copy of a SpinNP interface specification document so that multiple versions of the protocol can be specified by the same document. This gives a clear indication of the development of the protocol, and ties directly into the interface compiler, which generates marshalling and unmarshalling code that is aware of the differing contents of each version.

6 System Management

System management in the ONTAP GX system is responsible for maintaining the illusion that a GX cluster is a single system, even as new servers join the cluster, old servers are removed from the cluster, data is migrated between aggregates, and network addresses migrate between physical network cards.

Management is also responsible for ensuring that a large cluster hides the failures of internal components, so as to give the appearance not only of a reconfigurable cluster, but a cluster that never stops providing service.

The rest of this section describes the details of GX system management.

6.1 Replicated Database

The heart of the system management software is the replicated database, RDB. While RDB (not to be confused with Oracle s RDB) is mostly a new design, it uses some ideas from the "ubik" library used by AFS.

Databases built atop RDB store all cluster wide configuration information, including the location of all the cluster s virtual and physical resources, and the state of any long-running tasks that need to survive the failure of any given node. Each replica of an RDB database is guaranteed to be equivalent.

RDB provides two significant types of functionality.

First, it provides an election mechanism, used to elect a coordinator for updates to the underlying database. All updates to the database are funneled through this elected master, allowing a relatively straightforward implementation of the database. In addition, the elected master can perform application specific tasks that require at most a single instance to run successfully. For example, the elected master for the virtual interface manager also runs the single instance of the task that moves virtual interfaces between physical network cards in the event of a link or node failure in the cluster.

Because a new election is performed after a node failure, RDB can keep a database application running in the face of multiple node failures.

Second, RDB provides a transactional database facility allowing structured records to be atomically added to any system management databases. RDB databases are replicated among a set of machines in the cluster (referred to as a "ring). Currently, all machines in the cluster are in the RDB ring, and updates can be performed to the database as long as a majority of the servers in the ring are operational. A proper majority is required to unambiguously resolve the case where a network partitions. As RDB transactions are created, they contact the master for an (epoch, transaction ID) pair. The transaction ID is essentially a transaction count since the last time the cluster changed masters. The epoch is a sequence number of the number of times the master changed. Thus, by sorting on epoch and then ID, we get a global ordering of transactions in a cluster.

GX and RDB do not guarantee that the state of a database will be the same for the duration of an operation on a cluster. For example if a volume is one D-blade when a CIFS request arrives to file on the volume, the volume could move when the N-blade issues a SPINNP request. The N-blade would get an error indicating the volume has moved, and re-consult RDB for the volume s new D-blade.

A number of the databases built on top of RDB are discussed below.

6.2 Volume Location Database

The volume location database (VLDB) is an RDB replicated database. The VLDB stores a number of tables used by N-blades and the system management processes to locate in several steps - the D-blade corresponding to a particular volume. First, the volume is mapped to the aggregate that holds the volume. Second, because ownership of entire aggregates can be passed from one system to another, as in the case of system failure, the aggregate ID is mapped to a D-blade ID. Third, the D-blade ID is mapped to its network addresses via another RDB-based database, part of the VIF manager, described below. In addition, if a junction is encountered, the N-blade consults the junction table (kept in RDB, as described in section 4), to find the mounted child volume. Note that all of these mappings are composed and cached in the N-blade s memory, so that the results of all four lookups are typically available after a single hash table lookup.

The contents of each of these tables change during certain operations. When a volume is moved between servers, or between aggregates on the same server, the volume is first moved, and at the very end, the VLDB is updated to indicate that the volume has a new home aggregate. If a server in a storage failover partnership (where two D-blades share access to same set of disks and can takeover from each other in event of failure) fails, the surviving server updates the VLDB s aggregate to D-blade mapping, so that all of the N-blades in the system know where to find the aggregate. And if a system administrator changes the network address of an N-blade or D-blade, the blade s cluster IP address set is updated in the VIF manager s database as well.

6.3 Virtual Interface Location Database

The Virtual Interface (VIF) location database is managed by an active RDB process called the Virtual Interface manager (the VIF manager). The VIF manager is responsible for tracking which virtual network interfaces are associated with which physical network cards in the cluster, and ensuring that, should a server fail, that the IP addresses associated with the failed VIF are moved to an acceptable backup network card.

It is obviously important – regardless of what type of failures occur – that a given VIF is exported by no more than one physical network card at any instant. To provide this guarantee, VIFs are divided into two classes, fixed VIFs, which never migrate to other systems, even upon link or host failure, and moveable VIFs, which can be moved between servers in a cluster. If a cluster loses more than half of its servers, the RDB database will be unable to elect a master, and every VIF manager will see, after a small timeout, that it is no longer in contact with the RDB master for this database. When this occurs, all moveable VIFs are torn down, since the VIF manager can not distinguish between a majority of the servers in a cluster being down, and a network partition, where the master VIF manager process is simply no longer reachable from the local VIF manager. In the latter case, however, the VIF manager will likely reassign the moveable VIFs to another system, and to avoid having a cluster with duplicate IP addresses from different ports, a VIF manager that can t contact the VIF manager RDB master must tear down its moveable VIFs. Because fixed VIFs can never migrate to other servers, these VIFs can continue operation even after loss of contact with the RDB master, and indeed, this is the sole reason for the distinction. Of course, it is very rare for a cluster to lose more than half of its servers; typical failure modes are single server failures and network partitions, where a majority of servers remain in contact with each other.

6.4 System Management Framework

The system management framework provides administrative interfaces to the cluster. It is based upon the eponymously named SMF from Marconi Corporation. The system management framework generates a web user interface, command line interface and SNMP tables based upon a set of tables provided as input.

The commands invoked by this framework update configuration state stored in RDB, and thus mirrored to all servers in the cluster.

6.5 Job Controller

The job controller module is a part of the system management framework that provides support for long running operations that need to be restarted or cleaned up after a server failure.

The job controller provides RDB-based stable storage to keep track of the progress of a job. For example, a volume move operation is a complex multi-step operation with significant temporary state that must be cleaned up in the event of an error. A volume move begins by creating a volume on the destination server and creating a snapshot at the source server. The contents of the snapshot are then propagated to the empty volume at the destination. The snapshot propagation cycle is repeated several times and finally the VLDB is updated to point to the volume s new location. With this implementation, it is clear that if a crash occurs during the move, significant clean up must be performed before the operation can be restarted. The job controller provides a straightforward mechanism to ensure that the cleanup operations are invoked and the job restarted.

7 False Starts

The original design of junctions included the MSID of the child volume. This would have complicated data protection logic. For example, a volume might be backed up, and then the volume or subset thereof restored in a different place in namespace. Without additional logic, the restored volume would contain junctions pointing to child volumes that existed in other parts of the namespace.

The SPINNP protocol conveys semantics in a file access protocol independent manner. In most cases, the cost of this generality is nominal. However, when the D-blade converted READDIR results from WAFL to SPINNP, and the N-blade converted SPINNP READDIR results to NFS READDIR results, the CPU overhead was significant. We extended SPINNP to support NFS formatted READDIR results, and gained about 5% throughput using the benchmark described in the next section.

8 Performance

8.1 Overview

The system's performance characteristics derive from the division of the file service process into separate protocol termination and disk service modules (the N-blade and D-blade modules, respectively). As described in section 3.1, the high performance path consists of client requests traversing an N-blade routing across the cluster network and terminating at the appropriate D-blade (as illustrated in Figure 2).

The resources that drive performance include the clients networks, N-blade CPU and memory bandwidth, cluster network, D-blade CPU and memory bandwidth, and disk bandwidth. For typical NAS access patterns, we find the CPU balances across the N- and D-blades. A zero-copy networking stack removes the memory bandwidth bottleneck. Disk subsystem delays are ameliorated by adequate memory cache, including NVRAM to minimize latency of writes, on the D-blade.

8.2 Scaling

From a data traffic perspective, each N-blade is acting as no more than a switch to one of several other D-blades, which simplifies the analysis of the scaling limits of GX.

The performance of a node in a cluster (in operations per second) will be:

$$\begin{aligned} &\mathit{performance\_of\_local\_operations} \\ &+ \mathit{performance\_of\_remote\_operations} \end{aligned}$$

Where **performance_of_local_operations** is the performance of operations received by an N-blade that go to the D-blade that exists on the same node, and **performance_of_remote_operations** is the performance of operations received by the N-blade that go to D-blades that are not one the same node as the N-blade.

Assuming each N-blade receives an equal share of load from external clients, and assuming each N-blade evenly switches its received load to each D-blade, then among the D-blades, each D-blade is processing the same load. Let **P** be the performance of a single node cluster. Then in an **N** node cluster $performance\_of\_local\_operations = P / N$, and $performance\_of\_remote\_operations = P \times (N-1) / N \times E$, where **E** is the performance efficiency of remote operations. Thus the expected performance of a cluster of N nodes is:

$$\begin{aligned} &\text{expected\_performance\_cluster}(N) \\ &= N \times \left(\left(\frac{P}{N} \right) + \left(P \times \frac{N-1}{N} \times E \right) \right) = P + P \times (N-1) \times E \\ &= P \times (1 + (N-1) \times E) \end{aligned}$$

(For N > 0)

8.2.1 Scaling Results

To measure scaling, we used the SPEC SFS91_R1 V3.0 benchmark[*] to generate a standard workload of NFS (version 3) operations/second [Capps]. We present SFS numbers, because the SFS workload has a mix of I/O and metadata operations (both modifying and non-modifying) and is derived from some real customer workloads. SFS has uniform access rules that require no partitioning of data among the devices that comprise the system under test. Thus the SFS matches the assumptions listed in the previous section.

We rarely run large cluster performance benchmarks because of the labor and capital costs and because we ve found that setting up a two-node cluster, and configuring the benchmark to direct 100% of each N-blade s SpinNP traffic to the other (remote) D-blade is a sufficient predictor of how the large cluster will scale. As a result, we have just two large cluster figures to offer.

On a single mid range cluster node we achieved about 20,900 operations/second. On the two node ”100% remote cluster we achieved about 17,900 operations/second. Note that neither of these runs were compliant from SPEC s run rules, because the goal was to determine the maximum throughput, not to produce a compliant run. Thus, fewer than the required 10 ”load points were attempted with each single node run (see page 55 of [SPEC]). From the single node and ”100% remote runs we derive an expected efficiency of 85.6%. A 20 node cluster achieved about 318,000 operations/second versus an **expected_performance_cluster(20)** value of 360,000 operations/second. We did not have the same number of disk drives per node in the 20 node cluster as in the one and two node cluster, and believe that accounts for the discrepancy. We did not re-run with more drives because we were aiming for a much higher number.

Using our ”high end cluster node, a 24 node cluster configuration measured 1,032,461 SPECsfs97_R1.v3 operations per second, (with a corresponding overall response time of 1.53 milliseconds; for a definition of overall response time, see pages 55-56 of [SPEC]). Each node had three one gigabit/second Ethernet controllers, one for handling client traffic, two for the cluster interconnect. We do not have single node and 100% remote two cluster node numbers for the same software version on the high end node. However, several months after the 24 node run was published at SPEC s web site, we measured the high end single node at about 55,000 operations/second, and each node of the 100% remote two-node cluster at about 41,000 operations/second per node. The expected efficiency is about 75%, and **expected_performance_cluster(24)** = 1,003,750 or below 2.8% of actual results.

8.3 Load Balancing

In any collection of systems intended to service a common pool of clients, balancing load across the collection is an important consideration. We discuss two techniques used to address the problem.

8.3.1 Rebalancing Load

A fallacy of the previous section is the assumption that loads to the cluster or inside the cluster will be balanced. Because hot spots are a reality, GX offers two axes for balancing load.

The first axis is the ability to transparently migrate volumes among D-blades. With this capability, a given D-blade, and indeed a given set of disks, need not be a hot spot. Since volumes can be made arbitrarily small (while still being dynamically expandable), then, subject to the maximum file size needs of the application, one can create many small volumes that can be independently moved around the cluster as frequently as needed. The smaller the average volume size, the faster it can be moved in reaction to a load

[*] SPEC ™ and the benchmark name SPECsfs97_R1 ™ are registered trademarks of the Standard Performance Evaluation Corporation. For the latest SPECsfs97_R1 benchmark results visit www.spec.org (or more specifically: www.spec.org/osg/sfs97_R1).

imbalance and the easier to find a D-blade with sufficient spare cycles to service it.

The second axis is the ability to transparently migrate virtual network interfaces (VIFs) from one N-blade to another. The more VIFs a cluster has, the easier it is balance load across N-blades. Each N-blade has multiple VIFs, perhaps even multiple VIFs that refer to the same namespace. As the number of VIFs approaches the number of external clients, or the even the number of unique client source network addresses (for clients with multiple interfaces), the flexibility to respond to clients that dominate the capacity of a single N-blade improves. The clients that are using a VIF on an N-blade can be migrated by moving the VIF to an N-blade that has less load.

8.3.2 Load Balancing Mirrors

The root volume of a namespace, if not mirrored, becomes a performance bottleneck even if the traffic is mostly LOOKUP operations. To prevent the D-blade that holds the root volume of the namespace from being a bottleneck, a best practice for GX is to have a read-only load balancing (LB) mirror of the root volume of the namespace on each D-blade of the cluster. Each mirror is created via an asynchronous volume SnapMirror operation, which is similar to the asynchronous VSM feature in ONTAP-7G [Patterson]. This way, any read-only external client request (CIFS, NFS, etc.) that accesses the root volume can be serviced from any D-blade. Since external client requests arrive at N-blades, and since each N-blade will tend to have a local D-blade on the same node, the external request can be serviced locally, rather than being serviced by a D-blade that is on another node.

The set of load balancing mirrors, plus the writeable master volume, is collectively called a **volume family**. Each member of a volume family has a unique data set identifier (DSID), but each member shares the same MSID. This way, NFS requests can be routed to any available load balancing mirror for an MSID that appears in the request's NFS filehandle.

If a volume is a load balancing mirror, clients are permitted to access and modify the writeable master, but only if the access path is prefixed by the component "/.admin". Note that whether a volume in the path has a load balancing mirror is immaterial; the writeable version of the volume is always accessed. The ".admin" component is not quite a magic directory like WAFL's ".snapshot":

- It only appears at the root of the namespace.
- For NFS, only mount operations see it; NFS file and lock operations do not.

Thus if an NFS client mounts "/", and then tries to "cd" to ".admin" the LOOKUP will fail. Whereas, if the NFS client mounts a path starting with "/.admin", all access will be via the writeable masters for each volume family. GX accomplishes this by using one bit in the NFS file handle to indicate whether the writeable tree prefixed by "/.admin" is to be used or not.

Even though ".admin" appears at the root of the namespace this does not mean only the root volume in a namespace can have load balancing mirrors. Any volume can have load balancing mirrors. The writeable master of a volume family is always reachable by a path that starts with "/.admin".

If a writeable master is modified, a client accessing the volume through the normal path will not see the update until the system administrator directs the propagation of the updates to the load balancing mirrors.

8.4 Cluster Expansion

Ultimately, even with a perfectly balanced load, the aggregate load can exceed the cluster's capabilities. GX allows one to add nodes to the cluster, and thus provide new, idle places, to which to migrate load.

Nonetheless, expansion and migration will not suffice for all types of loads. A single volume can potentially experience more demand than the load capability of a single D-blade. A future version of GX will allow volumes to stripe across D-blades.

The problem of a single client outstripping an N-blade, or an N-blade's physical network interfaces is harder to solve, and ultimately requires external clients both capable of network trunking, and knowing the striping in the case of striped volumes.

9 Experiences with GX

We summarize the experiences of three customers who use GX today.

The first customer a provider of computer generated special effects for motion pictures - found that GX and its predecessor from Spinnaker provided the only storage solution capable of supporting its rendering load. Via the transparent volume move feature, the customer rebalanced storage usage across its GX nodes, and achieved average storage capacity utilization of nearly 90%, adding storage as needed.

The second customer is a supercomputing center supporting the information technology needs of scientists and other academic researchers. The customer uses a six node GX cluster for storing results of workloads and for user home directories with data consisting of mainly small files. The cluster handles bursts of data of up to 600 megabytes/second of mostly NFS traffic with some CIFS.

The third customer is a semiconductor manufacturer with a four node cluster. A single attached client can achieve 250 megabytes/second reading or writing, and with a 75/25 % mix of read and write, a single client achieves 650 megabytes/second. From 224 clients, an aggregate 800 megabytes/second write speed was measured, and aggregate gigabyte/second read (direct I/O) speed was seen, and using the 75/25 % mix 1.2 gigabytes/second read/write speed was achieved.

10 Conclusions

Data ONTAP GX provides many of the best features of previous scalable namespace file servers. It does this through widely deployed network file system protocols, including NFS and CIFS, avoiding changes to client software. The architecture translates file access requests to a common backend protocol that is used for all high-traffic communication within the cluster. This simplifies the implementation of the backend data server modules. The entire cluster is administered through a single interface, and administration is implemented through a number of cluster services. Performance scalability is linear, achieving a rate of over one million NFS operations per second on a 24 node cluster.

11 Acknowledgements

We thank Rich Sanzi, Darren Sawyer, Margo Seltzer, and five anonymous reviewers (provided by USENIX) for their constructive critiques of several drafts of this paper; and Dennis Chapman and Tianyu Jiang for information on customer experiences.

12 References

[Anderson] Anderson, D. et al, "Interposed Request Routing for Scalable Network Storage ACM Transactions on Computer Systems, vol 20, no 1, Feb 2002

[Campbell] Campbell, R., "Managing AFS: Andrew File System, ISBN 0-13-802729-3, 1998.

[Capps] Capps, D., "What s new in SFS 3.0 , NFS Conference, 2001.

[Hitz] Hitz, D. et al, "File System Design for an NFS File Server Appliance, USENIX Conference Proceedings, 1994.

[Howard] Howard, J. et al, "Scale and Performance in a Distributed File System, ACM Transactions on Computer Systems, Vol. 6, No. 1, 1988.

[Kazar1990] Kazar, M. et al, "DEcorum File System Architectural Overview, USENIX Conference Proceedings, 1990.

[Kazar2002] Kazar, M., "High Performance and Distributed NAS Server Architecture for Scalable and Global NFS file systems, NFS Industry Conference, 2002.

[Kleiman] Kleiman, S., "Vnodes: An Architecture for Multiple File System Types in UNIX," Proceedings of the USENIX Conference, 1986.

[Lamping] Lamping, U. et al, "Ethereal User s Guide, http://www.ethereal.com, 2005.

[Linn] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1 , RFC 2743, Internet Engineering Task Force, 2000.

[NetApp] Network Appliance, Inc., "Introduction to Data ONTAP™ 7G, TR 3356, 2005.

[Neuman] Neuman C., "The Kerberos Network Authentication Service (V5) , RFC 4120, Internet Engineering Task Force, 2005.

[Patterson] Patterson, H. et al, "SnapMirror®: File System Based Asynchronous Mirroring for Disaster Recovery, USENIX Conference on File and Storage Technologies Proceedings, 2002.

[SPEC] Standard Performance Evaluation Corporation, "SFS 3.0 Documentation Version 1.1, 2001.

[SNIA] "Common Internet File System Technical Reference, Storage Networking Industry Association, 2002.

[Sun] Sun Microsystems, "NFS: Network File System Protocol Specification, RFC 1094, Internet Engineering Task Force, 1989.

[Thekkath] Thekkath, C. et al, "Frangipani: a scalable distributed file system, Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, 1997.

//TRACE: Parallel trace replay with approximate causal events

Michael P. Mesnier*, Matthew Wachs, Raja R. Sambasivan, Julio Lopez,
James Hendricks, Gregory R. Ganger, David O'Hallaron

Carnegie Mellon University

Abstract

//TRACE[1] is a new approach for extracting and replaying traces of parallel applications to recreate their I/O behavior. Its tracing engine automatically discovers inter-node data dependencies and inter-I/O compute times for each node (process) in an application. This information is reflected in per-node annotated I/O traces. Such annotation allows a parallel replayer to closely mimic the behavior of a traced application across a variety of storage systems. When compared to other replay mechanisms, //TRACE offers significant gains in replay accuracy. Overall, the average replay error for the parallel applications evaluated in this paper is below 6%.

1 Introduction

I/O traces play a critical role in storage systems evaluation. They are captured through a variety of mechanisms [3, 4, 7, 16, 24, 50], analyzed to understand the characteristics and demands of different applications, and replayed against real and simulated storage systems to recreate representative workloads. Often, traces are much easier to work with than actual applications, particularly when the applications are complex to configure and run, or involve confidential data or algorithms.

However, one well-known problem with trace replay is the lack of appropriate feedback between storage response times and the arrival rate of requests. In most systems, storage system performance affects how quickly an application issues I/O. That is, the speed of a storage system in part determines the speed of the application. Unfortunately, information regarding such feedback is rarely present in I/O traces, leaving replayers with little guidance as to the proper replay rate. As a result, some replayers use the traced inter-arrival times (i.e., timing-accurate), some adjust the traced times to approximate how a workload might scale, and some ignore the traced times in favor of an "as-fast-as-possible" (AFAP) replay. For many environments, none of these approaches is correct [17]. Worse, one rarely knows how incorrect.

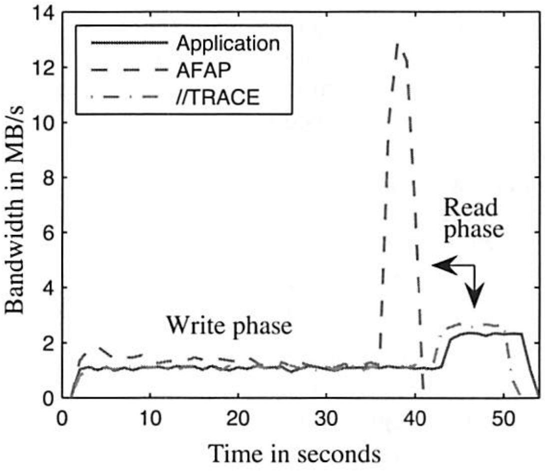


Figure 1: An example trace replay. This graph plots bandwidth over time, comparing an application to two different replayers. The application [28] simulates the checkpointing of a large-scale parallel scientific application. For this particular configuration, the write phase has numerous data dependencies (one outstanding I/O per process and a barrier synchronization after each write I/O) and the read phase is dominated by computation (processing of checkpoint data). AFAP replays the I/O traces "as-fast-as-possible," and //TRACE approximates both the synchronization and compute time. Because AFAP ignores synchronization and computation, it replays faster and places different demands on the storage system. //TRACE, however, closely tracks the application's I/O behavior.

Tracing and replaying *parallel* applications adds complexity to an already difficult problem. In particular, data dependencies among the compute nodes in a parallel application can further influence the I/O arrival rate and, therefore, its demands on a storage system. So, in addition to computation time and I/O time, nodes in a parallel application also have *synchronization* time; such is the case when, for example, one node's output is another node's input. If a replay of a parallel application is to behave like the real application, such dependencies must be respected. Otherwise, replay can result in unrealistic performance or even replay errors (e.g., reading a file before it is created). Figure 1 illustrates how synchronization and computation can affect the replay accuracy.

*Intel Research with Carnegie Mellon University

[1] Pronounced "*parallel trace*"

Parallel applications represent an important class of applications in scientific and business environments (e.g., oil/gas, nuclear science, bioinformatics, computational chemistry, ocean/atmosphere, and seismology). This paper presents //TRACE, an approach to accurately tracing and replaying their I/O in order to create representative workloads for storage systems evaluation.

//TRACE actively manages the nodes in a traced application in order to extract both the computation time and information regarding data dependencies. It does so in a black-box manner, requiring no modification to the application or storage system. An application is executed multiple times with artificial delays inserted into the I/O stream of a selected node (called the "throttled" node). Such delays expose data dependencies with other nodes and also assist in determining the computation time between I/Os. I/O traces can then be annotated with this information, allowing them to be replayed on a real storage system with appropriate feedback between the storage system and the I/O workload.

//TRACE includes an execution management script, a tracing engine, multi-trace post-processing tools, and a parallel trace replayer. Execution management consists of running an application multiple times, each time delaying I/O from a different node to expose I/O dependencies. The tracing engine interposes on C library calls from an unmodified application to capture I/O requests and responses. In the throttled node, this engine also delays I/O requests. The post-processing tools merge I/O traces from multiple runs and create per-node I/O traces that are annotated with synchronization and computation calls for replay. The parallel trace replayer launches a set of processes, each of which replays a trace from a given node by computing (via a tight loop that tracks the CPU counter), synchronizing (via explicit `SIGNAL()` and `WAIT()` calls), and issuing I/O requests as appropriate.

Experiments confirm that //TRACE can accurately recreate the I/O of a parallel application. For all applications evaluated in this paper, the average error is below 6%. Of course, the cost of //TRACE is the extra time required to extract the I/O dependencies. In the extreme, //TRACE could require *n* runs to trace an application executed on *n* nodes. Further, each of these runs will be slower than normal because of the inserted I/O delays. Fortunately, one can sample which nodes to throttle and which I/Os to delay, thus introducing a useful trade-off between tracing time and replay accuracy. For example, when tracing a run of Quake [2] (earthquake simulation), delaying every 10 I/Os (an I/O sampling period of 10) increases tracing time by a factor of 5 and yields a replay accuracy of 7%. However, one can increase the period to 100 for an 18% error and a tracing time increase of 1.7x.

This paper is organized as follows. Section 2 provides more background, motivates the design of //TRACE, and discusses the types of parallel applications for which it is intended. Section 3 overviews the design of //TRACE. Section 4 details the design and implementation of //TRACE. Section 5 evaluates //TRACE. Section 6 summarizes related work. Section 7 concludes.

2 Background & motivation

Storage system performance is critical for parallel applications that access large amounts of data. Of course, the most accurate means of evaluating a storage system is to run an application and measure its performance. However, taking such a "test drive" prior to making a design or purchase decision is not always feasible. Consequently, the industry has relied on a wide variety of I/O benchmarks (e.g., TPC benchmarks [46], Postmark [25], IOzone [31], Bonnie [8], SPC [42], SPECsfs [41], and Iometer [23]), many of which are even self-scaling [13] and adjust with the speed of the storage system. Unfortunately, while benchmarks are excellent tools for debugging and stress testing, using them to predict real world performance can be challenging; they can also be complex to configure and run [39]. In some cases, this has led to the creation of *pseudo-applications* — benchmarks crafted to reproduce the I/O activity of particular applications [28]. Unfortunately, designing a pseudo-application requires considerable expertise and knowledge of the real application, making them rare.

Trace replay provides an alternative to benchmarks and pseudo-applications: given a trace of I/O from a given application, a replayer can read the trace and issue the same I/O. The advantages of traces are their representativeness of real applications and their ease of use (applications can be difficult to configure or may even be confidential). Unfortunately, existing tracing mechanisms do not identify data dependencies across nodes (processes), making accurate parallel trace replay difficult.

In general, the rate at which each node in a parallel application issues I/O is influenced by its synchronization with other nodes (its data dependencies) and the speed of the storage system. In addition, the computation a node performs between I/Os will determine the maximum I/O rate. Unless I/O time, synchronization time, and compute time are all considered, the I/O replay rate may differ substantially from that of the application.

This work explores a new approach to trace collection and replay: a parallel trace replayer that issues the same I/O as the traced application and approximates its inter-I/O computation and data dependencies. In short, it tries to behave just like the application.

2.1 Trace replay models

There are two common models for trace replay: closed and open. In a *closed* model, I/O arrivals are dependent on I/O completions. In an *open* model, they are not [40]. In a closed model, the replay rate is determined by the *think time* between I/Os and the *service time* of each I/O in the storage system. The faster the storage system completes the I/O, the faster the next one will be issued, until think time is the limiting factor. In an open model, the replay rate is unaffected by the storage system.

When viewed from the perspective of a storage system, most I/O falls somewhere in between an open and closed model [17]. This is particularly the case when file systems and other middleware (e.g., caches) modulate an application's I/O rate. However, when viewed from the perspective of the application, the model is often a closed one (i.e., a certain number of outstanding I/O requests with a certain think time between I/Os). Therefore, as long as the traces are captured above the caches of the file and storage systems of interest (i.e., file-level as opposed to block-level), one can replay an application's file I/O using a closed model in order to create the same feedback as the traced application. The key challenge is determining what portion of the think time is constant and what portion will vary across storage systems.

For parallel applications, there are two components to think time: compute time and synchronization time. Compute time is that spent executing application code and, for the purposes of storage system evaluation, can be held constant during replay. Synchronization time, however, is variable — it represents time spent waiting on other nodes because of a data dependency and can therefore vary based on the rates of progress of the nodes.

2.2 Synchronization and the effect on I/O

A variety of synchronization mechanisms are in use today, including standard operating system mechanisms (signals, pipes, lock files, memory-mapped I/O) [35], message passing [20], shared memory [11], and remote procedure calls [44]. Also, some applications use hybrid approaches [34] (e.g., shared memory together with message passing). Although many of these mechanisms can be traced with a conventional tracing tool (e.g., *ltrace, strace, mpitrace*), it is unclear how one could replay asynchronous communication (e.g., applications using *select* or *poll*) without a semantic understanding of the application. Such asynchronous operations are used extensively in parallel applications in order to overlap communication with computation. Further, some of these synchronization mechanisms (e.g., shared memory) are not traceable using conventional tracing software.

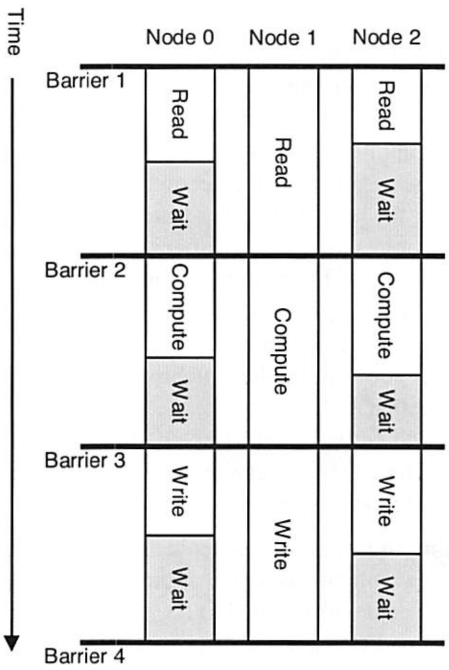


Figure 2: A hypothetical parallel application. All nodes are reading, modifying, and writing a shared data structure on disk, and barriers are used between each stage to keep the nodes synchronized. Node 1 happens to be the slowest node, forcing nodes 0 and 2 to wait. Note that under a different storage system, the I/O time for node 1 could change, thus resulting in changes in the synchronization time for nodes 0 and 2.

For these reasons, tracing and replaying synchronization calls is difficult. Namely, the variety of synchronization mechanisms and their semantics would need to be understood, determining causality for asynchronous messages would require application-level knowledge, and "untraceable" calls would not be easy to capture. Unfortunately, ignoring synchronization is not a viable option.

Consider Figure 2 which illustrates a hypothetical parallel application modifying a shared data structure; barriers [33] are used to keep the nodes synchronized between stages. As can be seen in the figure, the I/O time composes only a fraction of the overall running time; there is also compute time and synchronization ("wait") time. Moreover, if the speed of the storage system changes, the time each node spends waiting on other nodes could also change. These effects must be modeled during replay.

2.3 I/O throttling

The solution presented in this paper is motivated by the desire for a portable tracing tool that does not require knowledge of the application or the synchronization mechanisms being used. We accomplish this using a well-known technique called *I/O throttling* [9, 21].

Throttling involves selectively slowing down the I/O requests of an application, processing requests one at a time in the order they are received. In doing so, one can expose the data dependencies among the nodes in a parallel application. Consider the case where one node (node 0) writes a file that a second node (node 1) reads. To ensure the proper ordering (write followed by read), node 0 would signal node 1 after the file has been written. However, if I/O requests from node 0 are delayed, node 1 will block, waiting for the appropriate signal from node 0 (e.g., a remote procedure call). Although an I/O trace may not indicate the synchronization call, one can determine that node 1 is blocked (e.g., because there is no CPU or I/O activity) and conclude that it is blocked on node 0. The I/O traces can then be annotated to include this causal relationship between nodes 0 and 1.

Throttling I/O to expose dependencies and extract compute time is suitable for applications with compute nodes that produce deterministic I/O (i.e., for a given node, the sequence of I/O is the same for each run). For example, consider the case where n nodes write a file in a partitioned manner. That is, node 0 writes the first $1/n^{th}$ of the file, node 1 the second $1/n^{th}$, and so on. As such, although the global I/O scheduling can vary non-deterministically across multiple runs (e.g., due to process scheduling), the I/O issued by each node is fixed. For such applications, throttling will not change the I/O issued by a given node, the order in which a given node issues its I/O, or its data dependencies with other nodes; throttling only influences the timing. Although a variety of applications fit this model, we focus on parallel scientific applications [37], as they produce interesting mixes of computation, I/O, and synchronization.

3 Design overview

//TRACE discovers an application's data dependencies and compute time using I/O throttling. Summarizing from Section 2, the design requirements are as follows:

1. To adjust with the speed of the storage system, the traces must be replayed with a *closed* model.
2. To enforce data dependencies, the traces must be annotated with the inter-node synchronization calls.
3. To model computation, the inter-I/O compute time must be reflected in the traces.
4. To evaluate different file systems (e.g., log-structured vs. journaled) and different storage systems (e.g., blocks vs. objects [29]), the traces must be *file-level* traces, including all buffered and non-buffered synchronous POSIX [32] file I/O (e.g., `open`, `fopen`, `read`, `fread`, `write`, `fwrite`, `seek`).

//TRACE is both a tracing engine and a replayer, designed not to require semantic knowledge or instrumentation of the application or its synchronization mechanisms. The tracing engine, called the *causality engine*, is designed as a library interposer [14] (which uses the LD_PRELOAD mechanism) and is run on all nodes in a parallel application. The application does not need to be modified, but must be dynamically linked to the causality engine. Any shared library call issued by the application can be traced and optionally delayed using this mechanism.

The objectives of the causality engine are to intercept and trace the I/O calls, calculate the computation time between I/Os, and discover any causal relationships (i.e., the data dependencies) across the nodes. All of this information is stored in a per-node annotated I/O trace. A replayer (also distributed) can then mimic the behavior of the traced application, by replaying the I/O, the computation, and the synchronization. Although I/O calls to any shared library (e.g., MPI-IO, libc) can be traced and replayed, this work focuses on the POSIX I/O issued by an application through libc.

3.1 Discovering data dependencies

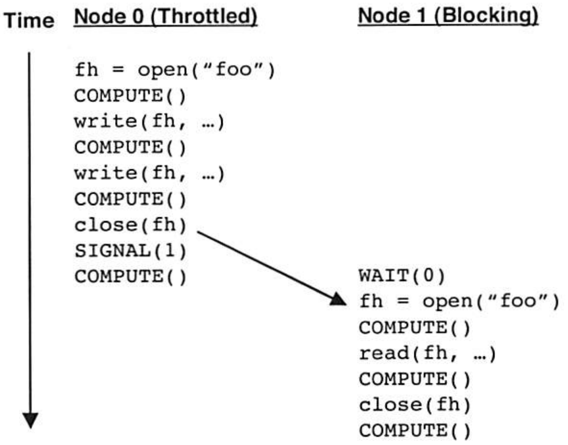
In general, one can automatically discover the data dependencies across all nodes by throttling each node in turn. When a node is being throttled, its I/O is delayed until all other nodes either exit or block/spin on an event. If a node exits, then it is obviously not dependent on the node being throttled. Conversely, any node that blocks must have some data dependency, perhaps only indirectly, with the throttled node. To reflect these dependencies, the throttled node will add a `SIGNAL()` to its trace and the blocking nodes will add a corresponding `WAIT()` to their traces. Figure 3 illustrates an example.

Of course, delaying every I/O could increase the running time of an application considerably. For this reason, one can selectively determine which I/Os to delay (I/O sampling) and which nodes to throttle (node sampling), thereby trading replay accuracy for tracing time. This trade-off is discussed further in Section 5.

3.2 Discovering compute time

In addition to discovering data dependencies, throttling assists in determining compute time. To determine the compute time, one must ensure that synchronization time is negligible or subtract the synchronization time from the think time. This paper discusses both approaches, but only the second is used in the evaluation.

Approach 1: The first approach recognizes that throttling a node makes its synchronization time negligible. When a node is being throttled, it is made to be slower



```
fh = open("foo")
COMPUTE()
write(fh, ...)
COMPUTE()
write(fh, ...)
COMPUTE()
close(fh)
SIGNAL(1)
COMPUTE()
                         WAIT(0)
                         fh = open("foo")
                         COMPUTE()
                         read(fh, ...)
                         COMPUTE()
                         close(fh)
                         COMPUTE()
```

Figure 3: Example of trace annotation. In this example, node 0 is writing to a file that node 1 is reading. By delaying I/O on node 0, the dependency can be exposed. Node 1 will block, waiting on node 0 to signal (through one of a number of possible mechanisms) that the file has been closed. Once the dependency has been discovered, the I/O traces are annotated with `SIGNAL()` and `WAIT()` calls that can be replayed. In addition, computation time can be added as `COMPUTE()` calls.

than all other nodes so as to expose data dependencies. Consequently, the think time between I/Os is all computation (e.g., node 1 in Figure 2 does not have to wait on nodes 0 and 2, because node 1 is the slowest node). The primary advantage of this approach is that it can be used even if an application is using "untraceable" synchronization mechanisms such as shared memory. The disadvantage is that I/O sampling can affect the computation calculation. This is discussed more in Section 5.

Approach 2: The second approach recognizes that many synchronization mechanisms are interrupt driven and rely on library or system calls for synchronization (e.g., a node may block while reading a socket). Therefore, given a list of calls that can potentially block, one can interpose on and calculate the time spent in each call, and then subtract this from the think time. Such an approach does not require a semantic understanding of any of the synchronization calls. Of course, this approach only works for synchronization mechanisms that issue library or system calls and will not work with applications that use "untraceable" synchronization (e.g., shared memory). Unlike the first approach, this one does not require a node to be throttled in order to extract computation, and the calculation is unaffected by I/O sampling.

Note, approaches 1 and 2 assume that multiple outstanding I/Os are achieved via multiple threads, each issuing synchronous I/O. The causality engine treats threads as separate "nodes" and traces each independently.

3.3 Putting it all together

Trace collection is an iterative process, requiring that an application be run multiple times, each time choosing a different node to throttle. Then, given a collection of traces (one trace for each of *n* application threads), a distributed replayer (*n* replay threads, one per trace) can replay the I/O, including any inter-I/O computation and synchronization, against dummy data files. Figure 4 illustrates this high-level architecture.

4 Detailed design

This section discusses the design of the causality engine and trace replayer in greater detail.

4.1 The causality engine

There are two modes of operation for the causality engine: *throttled* mode and *unthrottled* mode. For each run of the application, exactly one node is in throttled mode; all others are unthrottled. In both modes, each I/O is intercepted by the causality engine and stored in a trace for the respective node. This trace includes the I/O operations and their arguments. A node in throttled mode creates an I/O trace annotated with computation time (using Approach 1 or 2 from Section 3.2) and the "signaling" information. A node in unthrottled mode creates an I/O trace annotated with the "waiting" information and also the computation information if Approach 2 is used.

After *m* runs of an application ($m \leq n$), each node has *m* traces that must be merged. At most one of the traces per node contains the I/O when that node is being throttled, including `SIGNAL()` and `COMPUTE()` calls; all other traces reflect the I/O when the node is in unthrottled mode, including `WAIT()` calls, and also `COMPUTE()` calls if Approach 2 is used. Note that regardless of the mode, the I/O in all traces for a particular node should be identical, as our assumption is a deterministic I/O workload. If the I/O being issued by the application changes, we can easily detect this and report an appropriate error (e.g., "attempt to trace a non-deterministic application").

4.1.1 Throttled mode

When a node is being throttled, up to three pieces of information are added to the trace for each I/O. First, the compute time since the last I/O is determined (using Approach 1 or 2) and a `COMPUTE(<seconds>)` call is added to the trace. Second, the I/O operation and its arguments are added. Third, signaling information is added, as per the I/O *sampling period*.

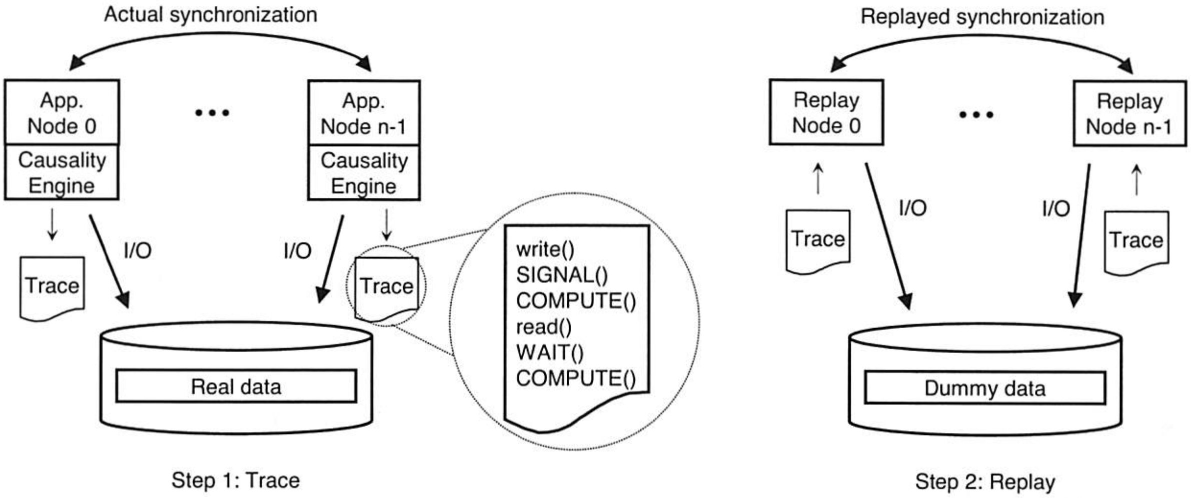


Figure 4: High-level architecture. While an application is running (left half of figure) the nodes are traced by the causality engine (a dynamically linked library) and selectively throttled to expose their data dependencies. Computation times are also estimated. This information is then used to annotate the I/O traces with `SIGNAL()`, `WAIT()` and `COMPUTE()` calls that can be easily replayed in a distributed replayer (right half of figure). During replay, dummy data files are use in place of the real data files.

The I/O sampling period determines how frequently the causality engine delays I/O to check for dependencies (e.g., a period of 1 indicates that every I/O is delayed) and therefore determines how many data dependencies are discovered. In general, if the sampling period is p, the causality engine will discover dependencies within p operations of the true dependency. Because the sampling period determines the rate of throttling, too large a sampling period can also affect the computation calculation. In these cases, Approach 2 (Section 3.2) is preferred.

When an I/O is being delayed, the causality engine delays issuing the I/O until all unthrottled nodes either exit or block (i.e., a dependency has been found). A remote procedure call is sent from the causality engine of the throttled node to a *watchdog* process on each unthrottled node to make this determination; some nodes may have exited, others may be blocked. If a node has exited, then it is not dependent on the delayed I/O. Otherwise, the throttled node adds a `SIGNAL(<unthrottled node id>)` to its trace, and the unthrottled node adds a corresponding `WAIT(<throttled node id>)` call to its trace. After the throttled node has received a reply from all of the watchdogs (one per unthrottled node), the I/O is issued. Algorithm 1 shows the pseudocode.

Of course, delaying I/O in this manner can produce indirect dependencies. For example, referring back to Figure 3, a sampling period of 1 will indicate that the `open()` call for node 1 is dependent on each I/O from node 0; namely, the `open()`, the two `write()` calls, and the `close()` — and the traces will be annotated as such to reflect this. However, the only signal needed is that following the `close()` operation, and the redundant `SIGNAL()` and `WAIT()` calls can be easily removed as a preprocessing step to trace replay. The indirect dependencies that cannot be removed are those due to transitive relationships. For example, if node 2 is dependent on node 1, and node 1 on node 0, the causality engine will detect the indirect dependency between nodes 0 and 2. Although these transitive dependencies add additional `SIGNAL()` and `WAIT()` calls to the traces, they never force a node to block unnecessarily.

As to selecting the proper sampling period, this depends on the application and storage system. Some workloads and storage systems may be more sensitive to changes in inter-node synchronization than others, so no one sampling period should be expected to work best for all. An iterative approach for determining the proper sampling period is presented in Section 5.

4.1.2 Unthrottled mode

When a node is being traced in unthrottled mode, up to three pieces of information are added to the trace for each I/O: a `COMPUTE()` call if Approach 2 is being used, the I/O operation and its arguments, and optionally a `WAIT()` call. The `WAIT()` is added by the watchdog process if it determines that an application node is blocked.

Recall (Algorithm 1), when the throttled node delays an I/O, it issues the `NodeIsBlocked()` call to each of the unthrottled nodes. The watchdog is responsible for handling this call. A node could block either in a system call (e.g., while reading a socket) or through user-level polling, and the watchdog should be able to handle both.

Algorithm 1: **ThrottledMode.** This function intercepts every I/O operation issued by the throttled node. First, the computation time since the previously issued I/O is added to the trace (Approach 1 shown). Computation time is simply the time since the last I/O completed. Second, the current I/O operation is added to the trace and optionally throttled as per the sampling period. If the I/O is throttled, the algorithm waits for all unthrottled nodes to block or complete execution. If a node is blocked, NodeIsBlocked() will return true, and a signal to that node will be added to the trace. Finally, the I/O is issued and the completion time is recorded.

```
AddComputeToTrace(GetTime()−previousTime);
AddOpToTrace();
if opCount is divisible by SAMPLE_PERIOD then
    foreach blocking node n do
        if NodeIsBlocked(n, thisNodeID) then
            AddSignalToTrace(n);
        endif
    endfch
endif
opCount ← opCount + 1;
IssueIO();
previousTime ← GetTime();
```

There are a variety of ways to determine if a node is blocked; the approach used by //TRACE is a simple one. Because blocking system calls used for inter-process synchronization (e.g., socket I/O, polling, select, pipes) can be intercepted by the causality engine, one can determine the time spent in each call. Similarly, if polling is used, the watchdog can just as easily determine the time spent computing (i.e., the time since the last I/O call completed). Therefore, to determine if an application is blocked, the watchdog checks with the causality engine (through shared memory) to see if the node is in a compute phase or in a system call. It then checks if the time spent in the compute phase or system call has exceeded a predetermined maximum; if so, it is blocked waiting on the throttled node. Note, this approach does not require a semantic understanding of any of the synchronization calls. Rather, the watchdog only needs to check that a computation phase or system call is not taking too long.

The maximum length of a computation phase or system call can be obtained from an analysis of an unthrottled run of the application (e.g., by using Unix *strace* to determine the maximum inter-arrival delay and system call time). These maxima must be chosen large enough to account for system variance. If too small a maximum is used, the watchdog may prematurely conclude that an application is blocked. In the best case, this introduces extra synchronization. In the worst case, it can lead to deadlock during replay. One heuristic used in this work is to increase the maximum system call time by a few factors. For example, if the maximum system call time in an unthrottled run of the application is 50 ms, then the maximum might be set to 100 ms; any system call taking longer than 100 ms is assumed to be blocked. Selecting too large a value only affects the trace extraction time.

```
---------       ---------       ---------
Trace 0.0       Trace 0.1       Trace 0.2
---------       ---------       ---------
read()          WAIT(1)         WAIT(2)
SIGNAL(1)       read()          read()
SIGNAL(2)
COMPUTE()
```

Figure 5: Before merging the traces. The application is such that node 0 waits for nodes 1 and 2 before issuing its `read()` and notifies nodes 1 and 2 after completing its `read()`. `Trace 0.0` shows the trace for node 0 when node 0 is being throttled. `Trace 0.1` is the trace for node 0 when node 1 is being throttled. And `Trace 0.2` is the trace for node 0 when node 2 is being throttled. Similar traces would exist for nodes 1 and 2.

4.2 Trace replay

Preparing traces for replay:

Following *m* runs of an application through the causality engine, each node has *m* traces that must be merged. All *m* traces for a given node should contain the same file I/O calls, otherwise an error will be flagged indicating that the application is not deterministic.

Recall that at most one of the *m* traces for a given node has the `SIGNAL()` calls for that node; this is the trace produced when the node is being throttled. The other traces for that node only have the `WAIT()` calls; these are the traces produced when other nodes are being throttled. After the merge, each I/O has at most $m-1$ preceding `WAIT()` calls, $m-1$ succeeding `SIGNAL()` calls, and one `COMPUTE()` call (obtained using Approaches 1 or 2).

The example in Figure 5 shows the trace files for a hypothetical 3-node application. In this case, every node is throttled in turn. Only the traces for node 0 are shown. A merge of these three traces will produce the final trace for node 0 (Figure 6).

Replaying the traces:

After traces have been annotated with `COMPUTE()`, `SIGNAL()`, and `WAIT()` calls, replay is straightforward, and the traces are easy to interpret. Each file operation can be replayed almost as-is; the syntax is similar to that

```
-------
Trace 0
-------
WAIT(1)
WAIT(2)
read()
SIGNAL(1)
SIGNAL(2)
COMPUTE()
```

Figure 6: After merging the trace files. `Trace 0.0`, `Trace 0.1`, and `Trace 0.2` are combined into one trace file for node 0. The merging process begins by creating a new trace file for node 0. For each I/O, all `WAIT()` calls are added first (the order does not matter), then the I/O call, then the `SIGNAL()` calls, and finally the `COMPUTE()`.

of Unix *strace*. Of course, filenames must be modified to point to dummy data files (which must be created prior to replay if they are not created by the application) and the replayers must maintain a mapping between the file handles in the trace and those assigned during replay. As for the synchronization, developers are free to implement these calls using any synchronization library (e.g., MPI [20], Java [19, 43], CORBA [49]) that is convenient (we use MPI); the `COMPUTE()` call is implemented by spinning for the specified amount of time. Computation is simulated by spinning, rather than sleeping, in order to induce a CPU load on the system like the application.

Figure 7 shows a merged trace file, obtained via the causality engine from a parallel scientific application [2]. In addition to enabling accurate replay, a trace instrumented with synchronization and computation reveals interesting information regarding program structure.

5 Evaluation

This work is motivated by four hypotheses.

Hypothesis 1 Data dependencies and computation must be independently modeled during replay, otherwise the replay may differ from the traced application.

Hypothesis 2 By throttling every node and delaying every I/O, the I/O dependencies and compute time can be discovered and accurately replayed.

Hypothesis 3 Not every I/O necessarily needs to be delayed in order to achieve good replay accuracy.

Hypothesis 4 Not every node necessarily needs to be throttled in order to achieve good replay accuracy.

```
/* barrier before opening output file */
WAIT(1)
WAIT(2)
SIGNAL(1)
SIGNAL(2)

/* open output file */
open64m("/pvfs2/output/mesh.e", 578, 416 ) = 17
COMPUTE(0.000148622)

/* write output file */
write(17, 4096) = 4096
COMPUTE(0.131106558)
_llseek(17, 8192, SEEK_SET) = 8192
COMPUTE(0.000000605)
write(17, 4096) = 4096
COMPUTE(0.000022173)
```

Figure 7: Example trace file. This is a snippet from a merged trace file for node 0 in a 3-node run of Quake, a parallel scientific application that simulates seismic events. The causality engine discovers that all nodes synchronize before opening and writing their output file (a mesh describing the forces during an earthquake). When replaying this trace, the open calls must be modified to point to dummy files that can be read and written. The replayer must maintain a mapping between the file handles in the trace (17 in this case) and those assigned during replay.

To test these hypotheses, three applications are traced and replayed across three different storage systems. The applications and storage systems chosen have different performance characteristics in order to highlight how application I/O rates scale (differently) across storage systems, and illustrate how //TRACE can collect traces on one storage system and accurately replay them on another. Recall, the primary goal of this work is to evaluate a new storage system, using trace replay to simulate the application. As such, traces are normally collected from one storage system and replayed on another.

There are three replay modes we could use as a baseline for comparison: a closed-loop as-fast-as-possible replay that ignores the think time between I/Os (AFAP), a closed-loop replay that replays think time (we call this *think-limited*), and an open-loop replay that issues I/O at the same time they are issued in the trace (timing-accurate [3]). Think-limited assumes that the think time (some combination of compute and synchronization) between I/Os is fixed. In general, we find think-limited to be more accurate than AFAP and therefore use it as our baseline comparator. A timing-accurate replay is not considered because, by definition, it will have an identical running time to the traced application. Note, a replayer that only models compute time (and ignores synchronization) requires some mechanism to distinguish compute time from synchronization time (e.g., a causality engine). Think-limited is therefore the best one can do before introducing such a mechanism.

Experiment 1 (Hypothesis 1) compares the running time of think-limited against the application. Because think-limited assumes a fixed synchronization time, one should expect high replay error when an application with significant synchronization time is traced on one storage system and replayed on another that has different performance.

Experiment 2 (Hypothesis 2) uses the causality engine to create annotated I/O traces. The traces are replayed and compared against think-limited.

Experiment 3 (Hypothesis 3) uses I/O sampling to explore the trade-off between tracing time and replay accuracy. Similarly, Experiment 4 (Hypothesis 4) uses node sampling to illustrate that not all nodes necessarily need to be throttled in order to achieve a good replay accuracy.

For all experiments, the traces used during replay are obtained from a storage system other than the one being evaluated. In other words, if storage system A is being evaluated, then the traces used for replay will have been collected on either storage system B or C. We report the error of the trace that produced the greatest replay error.

In all tests, running time is used to determine the replay accuracy, and the percent error is the evaluation metric. The reported errors are averages over at least 3 runs. More specifically, percent error is calculated as follows:

$$\frac{ApplicationTime - ReplayTime}{ApplicationTime} \times 100$$

Average bandwidth and throughput are not reported, as these are simply functions of the running time.

5.1 Experimental setup

Three parallel applications are used in the evaluation: `Pseudo`, `Fitness`, and `Quake`. All three applications use MPI [20] for synchronization (none use MPI-IO).

`Pseudo` is a pseudo-application from Los Alamos National Labs [28]. It simulates the defensive checkpointing process of a large-scale computation: MPI processes write a checkpoint file (with interleaved access), synchronize, and then read back the file. Optional flags specify whether or not nodes also synchronize after every write I/O, and if there is computation on the data between read I/Os. Three versions of the pseudo-application are evaluated: one without any flags specified (`Pseudo`), one with barrier synchronization (`PseudoSync`), and one with both synchronization and computation (`PseudoSyncDat`[2]).

`Fitness` is a parallel workload generator from Intel [22]. The generator is configured so that n MPI processes read non-overlapping portions of a file in turn; the first node reads its portion, then the second node, etc. There are only $n-1$ data dependencies: node 0 signaling node 1, node 1 signaling node 2, etc. This test illustrates a case where nodes are not proceeding strictly in parallel, but rather have some ordering that must be respected.

`Quake` is a parallel application developed at Carnegie Mellon University, used for simulating earthquakes [2]. It uses the finite element method to solve a set of partial differential equations that describe how seismic waves travel through the Earth (modeled as a mesh). The execution is divided into three phases. Phase 1 builds a multi-resolution mesh to model the region of ground under evaluation. The model, represented as an etree [47], is an on-disk database structure; the portion of the database accessed by each node depends on the region of the ground assigned to that node. Phase 2 writes the mesh structure to disk; node 0 collects the mesh data from all other nodes and performs the write. Phase 3 solves the equations to propagate the waves through time; computation is interleaved with the I/O, and the state of the simulated region is periodically written to disk by all nodes. `Quake` runs on a parallel file system (PVFS2 [10]) which is mounted on the storage system under evaluation.

The applications are traced and replayed on three storage systems. The storage systems are iSCSI [38] RAID arrays with different RAID levels and varying amounts of disk and cache space. Specifically, **VendorA** is a 14-disk (400GB 7K RPM Hitachi Deskstar SATA) RAID-50 array with 1GB of RAM; **VendorB** is a 6-disk (250GB 7K RPM Seagate Barracuda SATA) RAID-0 with 512 MB of RAM; and **VendorC** is an 8-disk (250GB 7K RPM Seagate Barracuda SATA) RAID-10 with 512 MB of RAM.

The applications and replayer are run on dedicated compute clusters. `Pseudo` and `Fitness` are run on Dell PowerEdge 650s (2.67 GHz Pentium 4, 1 GB RAM, GbE, Linux 2.6.12); `Fitness` is configured for 4 nodes, `Pseudo` is configured for 8. `Quake` is run on a cluster of Supermicros (3.0 GHz dual Pentium 4 Xeon, 2.0 GB RAM, GbE, Linux 2.6.12), and is configured for 8 nodes. The local disk is only used to store the trace files and the operating system. `Pseudo` and `Fitness` access the arrays in raw mode. For these applications, each machine in the cluster connects to the same array using an open source iSCSI driver [22]. For `Quake`, each node runs PVFS2 and connects to the same PVFS2 server, which connects to one of the storage arrays via iSCSI.

[2] We increased the computation after each read by a factor of 100 to make `PseudoSyncDat` significantly different than `PseudoSync`.

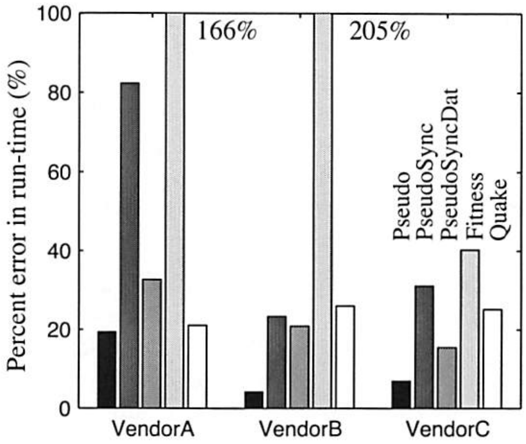


Figure 8: Think-limited error (Experiment 1). Think-limited is most accurate for Pseudo, which contains little synchronization. The other applications (PseudoSync, PseudoSyncDat, Fitness, and Quake) experience more error.

5.2 Experiment 1 (think-limited)

Think-limited replays the trace files against the storage devices with a fixed amount of think time between I/Os. The I/O traces are collected through the causality engine running in a special mode: no I/O is delayed and the COMPUTE() calls also include any synchronization time.

Figure 8 shows the replay error of think-limited. The best result is for Pseudo, which performs little synchronization (a single barrier between the write phase and read phase). The replay errors on the VendorA, VendorB, and VendorC, storage systems are, respectively, 19%, 4%, and 7% (i.e., the trace replay time is within 19% of the application running time across all storage systems). Unfortunately, it is only for applications such as these (i.e., few data dependencies) that think-limited does well.

Looking now at PseudoSync, one can see the effects of synchronization. All nodes write their checkpoints in lockstep, performing a barrier synchronization after every write I/O. The errors are 82%, 23%, and 31%, indicating that synchronization, when assumed to be fixed, can lead to significant replay error when traces collected from one storage system are replayed on another.

In PseudoSyncDat, nodes synchronize between I/Os and also perform computation. The errors are 33%, 21%, and 15%. In this case, adding computation makes the replay time less dependent on synchronization.

Fitness is a partitioned, read-only workload. Each node sequentially reads a 1 GB region of the disk, with no overlap among the nodes. The nodes proceed sequentially: node 0 reads its entire region first and then signals node 1, then node 1 reads its region and signals node 2, etc. Ignoring these data dependencies during replay will result in concurrent access from each node, which in this case increases performance on each storage system.[3] The replay errors are 166%, 205%, and 40%.

Quake represents a complex application with multiple I/O phases, each with a different mix of compute and synchronization. The think-limited replay errors for Quake are 21%, 26%, and 25%. As with the other applications tested, these errors in running time translate to larger errors in terms of bandwidth and throughput. For example, in the case of Quake, think-limited transfers the same data in 79%, 74%, and 75% of the time, resulting in bandwidth and throughput differences of 27%, 35%, and 33%, respectively. This places unrealistic demands on the storage system under evaluation.

5.3 Experiment 2 (I/O throttling)

Experiment two compares the accuracy of //TRACE and think-limited. Results are shown in Figure 9, which is the same as Figure 8, with //TRACE added for comparison.

//TRACE offers no significant improvement for Pseudo, and this result is expected given that Pseudo has few data dependencies. However, for both PseudoSync and PseudoSyncDat, //TRACE offers substantial gains. Namely, the maximum replay error is reduced from 82% to 17% for PseudoSync and 33% to 10% for PseudoSyncDat. These improvements are due to the replayed synchronization: a barrier after every write I/O, which //TRACE approximates with 8 SIGNAL() and 8 WAIT() calls per node (a barrier requires all nodes to signal and wait on all other nodes before proceeding).

Looking at Fitness, one sees even greater improvement. Namely, the maximum replay error is reduced from 205% to 5%. There are only 3 data dependencies approximated by //TRACE: node 0 signaling node 1 after it completes is read, 1 signaling 2, and 2 signaling 3. Nonetheless, these dependencies enforce a sequential execution of the I/O (which is what Fitness intended); when ignored, the result is concurrent access from all nodes (a different workload altogether). Therefore, it is not the number of data dependencies discovered that determines replay accuracy, but rather how these dependencies impact the storage system.

The Quake workload highlights how accurately //TRACE replays complex applications with multiple I/O phases, having different mixes of I/O, compute, and synchronization. Relative to think-limited, the maximum replay error is reduced from 26% to 8%.

[3] For storage devices with little cache space and no read-ahead, concurrent sequential read accesses can increase the number of seek operations and decrease performance.

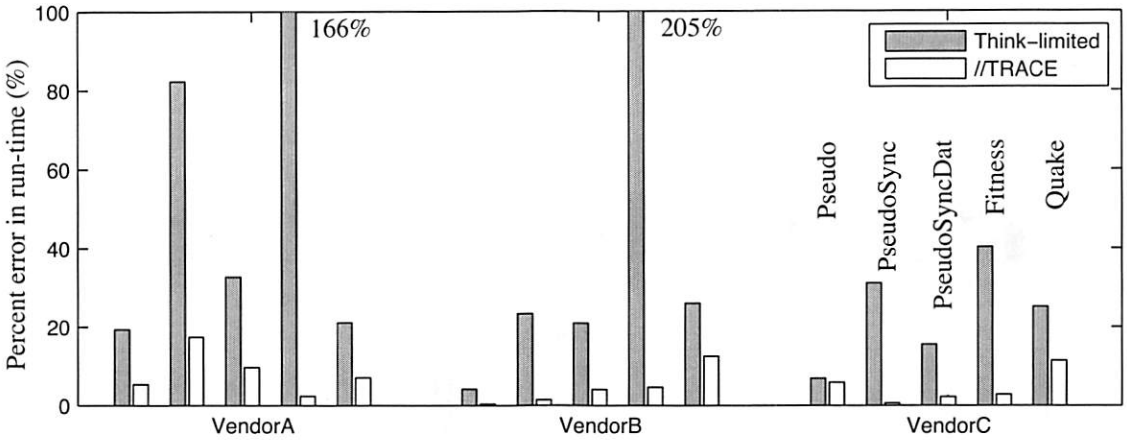


Figure 9: Think-limited vs. //TRACE (Experiment 2). The error incurred by //TRACE is less than think-limited across all applications and storage arrays. The improvement is most pronounced for the applications with large amounts of synchronization.

5.4 Experiment 3 (I/O sampling)

The causality engine can throttle every I/O issued by every node. However, sufficient replay accuracy can be obtained in significantly less time. In particular, one can sample across both dimensions (i.e., which I/Os to delay and which nodes to run in throttled mode). This experiment explores the first dimension, specifically the trade-off between replay accuracy and the I/O sampling period.

Five sampling periods are compared (1, 5, 10, 100, and 1000). As discussed in Section 3, the period determines the frequency with which I/O is delayed. If the sampling period is 1, every I/O is delayed. If the sampling period is 5, every 5^{th} I/O is delayed, etc. Given this, one would expect the sampling period to have the greatest impact on applications with a large number of I/O dependencies.

Note, I/O sampling can affect the computation calculation when using the throttling-based approach (Approach 1 in Section 3.2). Recall that throttling a node makes it slower than all the others. If the sampling frequency is too low (a large sampling period), then that node may not always be the slowest, thereby potentially introducing synchronization time into the trace which would be inadvertently counted as computation. Therefore, timing the system calls to determine computation time (Approach 2 in Section 3.2) is a more effective approach when using large sampling periods. None of the applications evaluated use "untraceable" mechanisms for synchronization, allowing Approach 2 to work effectively.

Figure 10 plots replay accuracy against the I/O sampling period, for each of the applications and storage systems being evaluated. Beginning with Pseudo, for which there are few data dependencies (i.e., all nodes must complete their last write before any node begins reading the checkpoint), one should expect little difference in replay error among the different sampling periods. As shown in the figure, the replay error for Pseudo is within 10% for all sampling periods and storage arrays.

PseudoSync and PseudoSyncDat behave quite differently (i.e., a barrier after every write I/O) and highlight the trade-off between tracing time and sampling period. As shown in the figure, replay error quickly decreases with smaller sampling periods. Notice the prominent staircase effect as the sampling is decreased from 1000 to 1. These applications represent the worst case scenario for sampling, where data dependencies are the primary factor influencing the replay rate.

Looking now at Fitness, one sees behavior very similar to Pseudo. Both have few data dependencies and do not require frequent I/O sampling for accurate replay. The error for Fitness is within 5% for all sampling periods.

Quake performance is influenced by synchronization (like PseudoSync and PseudoSyncDat). So, discovering more data dependencies offers improvements in replay accuracy. For example, an I/O sampling period of 5 yields a 2.9% replay error, compared to 21% error for an I/O sampling period of 1000.

5.4.1 I/O sampling discussion

To choose the "optimal" sampling period, one must consider both the application and the storage system. The only sampling period guaranteed to find all of the data dependencies, for arbitrary applications and storage systems, is a period of 1. Larger sampling periods may begin to introduce some amount of tracing error. The trade-off is replay accuracy for tracing time.

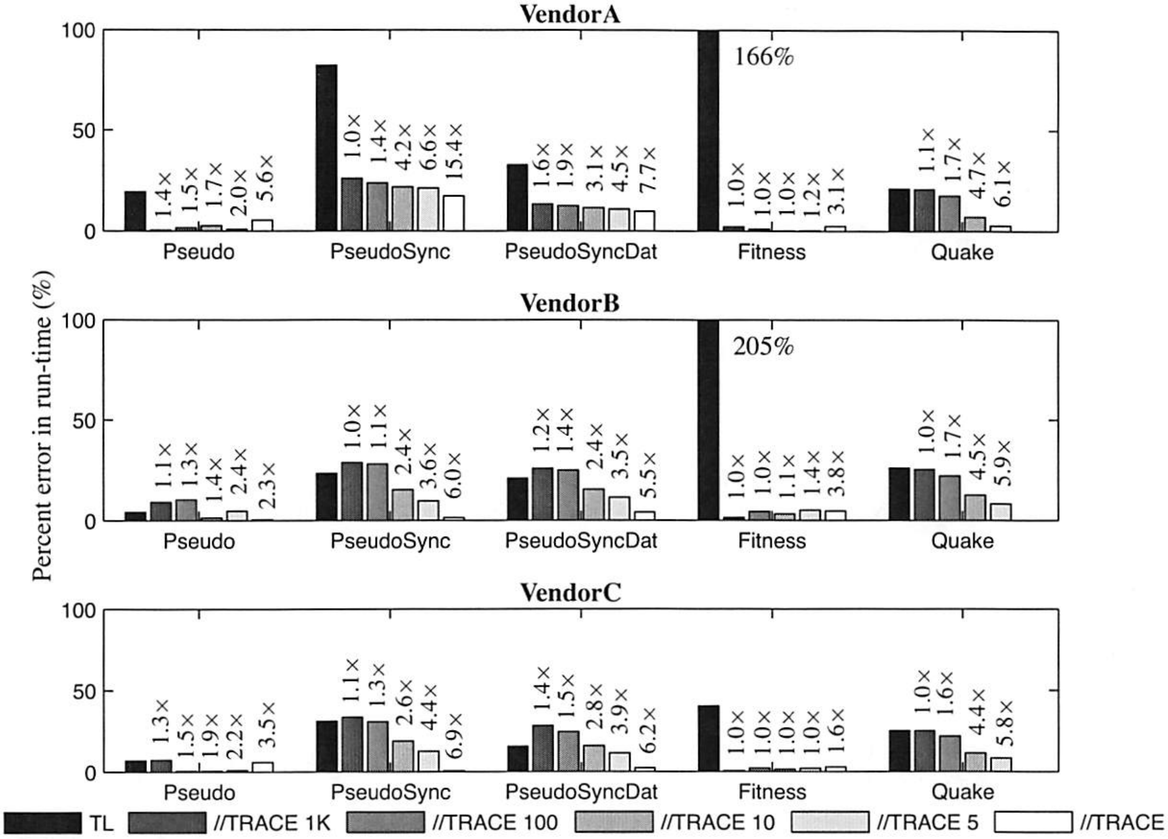


Figure 10: Sampling vs. accuracy trade-off (Experiment 3). By sampling which I/Os to delay, the tracing time can be reduced at the potential cost of replay accuracy. This graph plots the replay error over each application and storage system, for different sampling periods: 1000, 100, 10, 5, and 1. Think-limited (TL) is shown for comparison. The fractional increase in running time for the application is shown above each bar. These graphs illustrate the trade-off between tracing time and replay accuracy, but also show that larger sampling periods can achieve good replay accuracy, with minimal impact on the running time. (Note, the smallest sampling period shown for Quake is 5, as this period already produces only a 2.9% replay error.)

Intuitively, applications with a large number of data dependencies will realize longer tracing times as the data dependencies are being discovered by the causality engine. Recall from Section 4.1 that for every delayed I/O, the throttled node waits for all other nodes to block or complete execution, and the time for the watchdog to conclude that a node is blocked is derived from the expected maximum compute phase or system call time for that application node. Therefore, the tracing time can vary dramatically across applications and storage systems. Figure 10 shows the average increases in application running time for various I/O sampling periods. In the best case, I/O sampling introduces almost no overhead (a running time increase close to 1.0) and yields significantly better replay accuracy than think-limited (e.g., sampling every 1000 I/Os of `PseudoSync` reduces the error of think-limited by over a factor of 3 on VendorA, from 82% to 26%).

In practice, one can trace applications with a large sampling period (e.g., 1000) and work toward smaller sampling periods until a desired accuracy, or a limit on the acceptable tracing time, is reached. Of course, the "optimal" sampling period of an application when traced on one storage system may not be optimal for another. Therefore, one should replay a trace across a collection of different storage systems to help validate the accuracy of a given sampling period. We believe that developing heuristics for validating traces across different storage systems in order to determine a "globally optimal" sampling period is an interesting area for future research.

However, even with an optimally selected sampling period, an application is still run once for each application node in order to extract I/O dependencies. Therefore, node sampling (sampling which nodes to throttle) is necessary to further reduce the tracing time.

5.5 Experiment 4 (Node sampling)

This experiment shows that low replay error can be achieved without having to throttle every node. It compares the replay error for various values of m (the number of nodes throttled, chosen independently at random).

In some cases, node sampling can introduce error. Such is the case with `Fitness`, which only has 3 data dependencies. If any one of these is omitted, one of the nodes will issue I/O out of turn (resulting in concurrent access to the storage system). This represents a pathological case for node sampling. For example, when running on the VendorB platform, replay errors when throttling 1, 2, 3, and 4 nodes, are 37%, 29%, 17%, and 5%.

`Quake` and `PseudoSyncDat` are more typical applications. Figure 11 plots their error. With `Quake`, one achieves an error of 13% when throttling 2 of the 8 nodes (I/O sampling period of 5). Similarly, `PseudoSyncDat` achieves an 8% error when throttling 4 of the 8 nodes (I/O sampling period of 1). As with I/O sampling, one can sample nodes iteratively until a desired accuracy is achieved, and the traces can be evaluated across various storage systems to validate accuracy.

Interestingly, throttling more nodes does not necessarily improve replay accuracy (e.g., randomly throttling four nodes in `Quake` produces more error than throttling two). Because this experiment randomly selects the the throttled nodes, the sampled nodes may not necessarily be the ones with the most performance-affecting data dependencies. Therefore, heuristics for intelligent node sampling are required to more effectively guide the trace collection process and further reduce tracing time. In addition, learning to recognize common synchronization patterns (e.g,. barrier synchronization) could reduce the number of nodes that would need to be throttled. These are both interesting areas of future research.

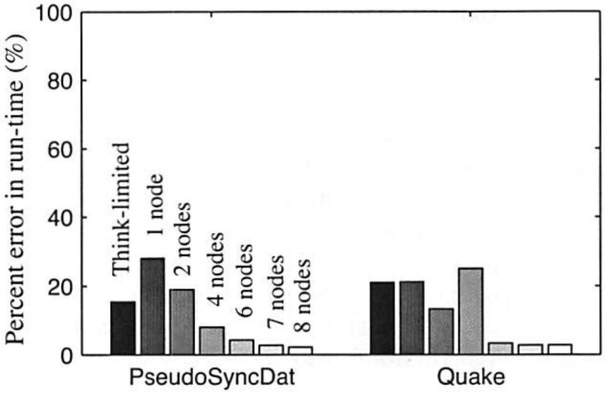


Figure 11: Node sampling (Experiment 4). Low replay error can be achieved without having to throttle every node. This graph plots the replay error for various values of m (number of nodes throttled). For `Quake`, error increases when sampling 4 nodes instead of 2, indicating that the nodes randomly selected for throttling determine the replay accuracy. (Not all storage systems are presented in this graph. The `PseudoSyncDat` results are from the VendorC array and `Quake` is from VendorA.)

6 Related work

A variety of tracing tools are available for characterizing workloads and evaluating storage systems [4, 7, 16, 24, 50]. However, these solutions assume no data dependencies, making accurate parallel trace replay difficult.

There are also a number of tools for tracing, replaying and debugging parallel applications [5, 15, 18, 26, 30, 36]. Because these tools are used to reduce the inherent non-determinism in message passing programs in order to make debugging easier (e.g., to catch race conditions or deadlock), they deterministically replay non-deterministic applications in order to produce the same set of events, and hence synchronization times, that occurred during the traced run. In contrast, the goal of //TRACE is to replay I/O traces so as to reproduce (realistically) any non-determinism in the the global ordering of I/O being issued by the compute nodes.

Throttling has been used successfully elsewhere to correlate events [9, 21]. By imposing variable delays in system components, one can confirm causal relationships and learn much about the internals of a complex distributed system. //TRACE follows this same philosophy, by delaying I/O at the system call level in order to expose the causal relationships among nodes in a parallel application; this information is then used to approximate the causal relationships during trace replay.

There are also black-box techniques for intelligently "guessing" causality, and these do not require throttling or perturbing the system. In particular, message-level traces can be correlated using signal processing techniques [1] and statistics [12]. The challenge is distinguishing causal relationships from coincidental ones.

Operating system events can be used to track the resource consumption of an application [6, 45] and also determine the dominant causal paths in a distributed system. Such "whitebox" techniques would complement //TRACE, especially when debugging the performance of a system, by providing detail as to the source of a data dependency. In addition, system call tracing has been successfully used to discover dependencies among processes and files for intrusion detection [27] and result caching [48].

7 Conclusion

This paper presents a technique for accurately extracting and replaying I/O traces from parallel applications. By selectively delaying I/O while tracing an application, computation time and inter-node dependencies can be discovered and approximated in trace annotations. Unlike previous approaches to trace collection and replay, such approximation allows a replayer to closely mimic the behavior of a parallel application. Across the applications and storage systems evaluated in this study, the average replay error is below 6%.

Acknowledgements

We thank our colleagues at Los Alamos National Laboratories (John Bent, Gary Grider and James Nunez) and Intel (Brock Taylor) for numerous discussions and valuable input. We thank the reviewers, in particular for their suggestion to include results for think-limited. We thank our shepherd, Kai Shen, for his additional feedback. We thank the members of the PDL Consortium (including APC, Cisco, EMC, Hewlett-Packard, Hitachi, IBM, Intel, Network Appliance, Oracle, Panasas, Seagate, and Symantec) for their interest, insights, feedback, and support; and we thank Intel, IBM, Network Appliance, Seagate, and Sun for hardware donations that enabled this work. This material is based on research sponsored in part by the NSF, via grants CNS-0326453, CNS-0509004, CCF-0621508, and IIS-0429334, by the Air Force Research Laboratory, under agreement F49620–01–1–0433, by the Army Research Office, under agreement DAAD19–02–1–0389, by a subcontract from the Southern California Earthquake Center's CME Project as part of NSF ITR EAR-01-22464, and by the Department of Energy under Award Number DE-FC02-06ER25767. Matthew Wachs is supported in part by an NDSEG Fellowship, sponsored by the Department of Defense. James Hendricks is supported in part by a NSF Graduate Research Fellowship.

References

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM Symposium on Operating System Principles* (Bolton Landing, NY, 19–22 October 2003), pages 74–89. ACM Press, 2003.

[2] Volkan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez, Omar Ghattas, Eui Joong Kim, Julio Lopez, David O'Hallaron, Tiankai Tu, and John Urbanic. High Resolution Forward and Inverse Earthquake Modeling on Terasacale Computers. *ACM International Conference on Supercomputing* (Phoenix, AZ, 15–21 November 2003), 2003.

[3] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: a toolkit for flexible and high fidelity I/O benchmarking. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004), pages 45–58. USENIX Association, 2004.

[4] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: a file system to trace them all. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004), pages 129–145. USENIX Association, 2004.

[5] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. TraceBack: first fault diagnosis by reconstruction of distributed control flow. *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, 11–15 June 2005), pages 201–212. ACM Press, 2005.

[6] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004). USENIX Association, 2004.

[7] Matt Blaze. NFS tracing by passive network monitoring. *USENIX.* (San Francisco), 20-24 January 1992.

[8] T. Bray. Bonnie benchmark, 1996. http://www.textuality.com.

[9] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environments. *7th International Symposium on Integrated Network Management* (Seattle, WA, 14–18 March 2001). IFIP/IEEE, 2001.

[10] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: a parallel file system for linux clusters. *Linux Showcase and Conference* (Atlanta, GA, 10–14 October 2000), pages 317–327. USENIX Association, 2000.

[11] Rohit Chandra. *Parallel Programming in OpenMP.* Morgan Kaufmann, October 2000.

[12] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. *Symposium on Networked Systems Design and Implementation* (San Francisco, CA, 29–31 March 2004), pages 309–322. USENIX Association, 2004.

[13] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation - Self-Scaling I/O Benchmarks, Predicted I/O Performance. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA, 10–14 May 1993). ACM, 1993.

[14] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. *Summer USENIX Technical Conference* (Boston, MA), pages 267–278, 6–10 June 1994.

[15] J. Chassin de Kergommeaux, M. Ronsse, and K. De Bosschere. Mpl*: efficient record/replay of nondeterministic features of messagepassing libraries. *In Proceedings of EuroPVM/MPI'99*, **1697**. Springer Verlag, Sep 1999.

[16] Daniel Ellard and Margo Seltzer. New NFS tracing tools and techniques for system analysis. *Systems Administration Conference* (San Diego, CA), pages 73–85. Usenix Association, 26–31 October 2003.

[17] Gregory R. Ganger and Yale N. Patt. Using system-level models to evaluate I/O subsystem designs. *IEEE Transactions on Computers*, **47**(6):667–678, June 1998.

[18] Dennis Geels, Gautum Altekar, Scott Shenker, and Ion Stoico. Replay debugging for distributed applications. *USENIX Annual Technical Conference* (Boston, MA, 30–03 June 2006), pages 289–300. USENIX Association, 2006.

[19] James Gosling and Henry McGilton. *The Java language environment.* Technical report. October 1995.

[20] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI, 2nd Edition.* MIT Press, November 1999.

[21] Haryadi S. Gunawi, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiri Schindler. Deconstructing commodity storage. *ACM International Symposium on Computer Architecture* (Madison, WI, 04–08 June 2005), pages 60–71. IEEE, 2005.

[22] Intel. Storage Toolkit. www.sourceforge.net/projects/intel-iscsi.

[23] Intel Corporation. IOmeter, 1998. http://www.iometer.org.

[24] Nikolai Joukov, Timothy Wong, and Erez Zadok. Accurate and efficient replaying of file system traces. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 336–350. USENIX Association, 2005.

[25] Jeffrey Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[26] R. Kilgore and C. Chase. Re-execution of distributed programs to detect bugs hidden by racing messages. *30th Hawaii International Conference on System Sciences*, Jan 1997.

[27] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Symposium on Operating System Principles* (Lake George, NY, 19–22 October 2003), pages 223–236. ACM, 2003.

[28] Los Alamos National Laboratory. Pseudo application. http//institute.lanl.gov/data.

[29] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based Storage. *Communications Magazine*, **41**(8):84–90. IEEE, August 2003.

[30] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *ACM International Conference on Supercomputing* (Minneapolis, Minnesota, November 1992), pages 502–511. Institute of Electrical Engineers Computer Society Press, November 1992.

[31] William D. Norcott. IOzone, http://www.iozone.org, 2001.

[32] IEEE Standards Project P1003.1. *Portable Operating System Interface (POSIX), Part 2: System Interfaces*, volume 2, number ISO/IEC 9945, IEEE Std 1003.1-2004. IEEE, 2004.

[33] Peter Pacheco. *Parallel Programming with MPI, 1st edition*. Morgan Kaufmann, October 1, 1996.

[34] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Science/Engineering/Math, 2004.

[35] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, September 2003.

[36] Michiel Ronsse, Koen De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. *Fourth International Workshop on Automated Debugging (AADEBUG 2000)* (Munich, Germany, August 2000). IRISA, 2000.

[37] Emilia Rosti, Giuseppe Serazzi, Evgenia Smirni, and Mark S. Squillante. Models of Parallel Applications with Large Computation and I/O Requirements. *Transactions on Software Engineering*, **28**(3):286–307. IEEE, March 2002.

[38] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). IETF, April 2004. http://www.ietf.org/rfc/rfc3720.txt.

[39] M. Satyanarayanan. Lies, Damn Lies and Benchmarks. *Hot Topics in Operating Systems* (Rio Rico, AZ, 29–30 March 1999). USENIX Association, 1999.

[40] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open Versus Closed: A Cautionary Tale. *Symposium on Networked Systems Design and Implementation* (San Jose, CA, 08–10 May 2006), pages 239–252. USENIX Association, 2006.

[41] Standard Performance Evaluation Corporation. SPEC SFS97 v3.0, December 1997. http://www.storageperformance.org.

[42] Storage Performance Council. SPC-2 Benchmark, December 2005. http://www.storageperformance.org.

[43] Mihai Surdeanu. Distributed Java virtual machine for message passing architectures. *International Conference on Distributed Computing Systems* (Taipei, Taiwan, 10–13 April 2000), pages 128–135. IEEE, 2000.

[44] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, January 2002.

[45] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: Tracking activity in a distributed storage system. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Saint-Malo, France, 26–30 June 2006), 2006.

[46] Transaction Processing Performance Council. Performance Benchmarks TPC-C and TPC-H. http://www.tpc.org.

[47] Tiankai Tu, David R. O'Hallaron, and Julio Lopez. Etree - a database-oriented method for generating large octree meshes. *Meshing Rountable* (Ithaca, NY, 15–18 September 2002), pages 127–138, 2003.

[48] Amin Vahdat and Thomas Anderson. Transparent Result Caching. *USENIX Annual Technical Conference* (New Orleans, LA, 15–19 June 1998). USENIX Association, 1998.

[49] Zhonghua Yang and Keith Duddy. CORBA: a platform for distributed object computing. *Operating Systems Review*, **30**(2):4–31, April 1996.

[50] Ningning Zhu, Jiawu Chen, and Tzi-Cker Chiueh. TBBT: scalable and accurate trace replay for file server evaluation. *Conference on File and Storage Technologies* (San Francisco, CA, 13–16 December 2005), pages 323–336. USENIX Association, 2005.

Karma: Know-it-All Replacement for a Multilevel cAche

Gala Yadgar
Computer Science Department, Technion

Michael Factor
IBM Haifa Research Laboratories

Assaf Schuster
Computer Science Department, Technion

Abstract

Multilevel caching, common in many storage configurations, introduces new challenges to traditional cache management: data must be kept in the appropriate cache and replication avoided across the various cache levels. Some existing solutions focus on avoiding replication across the levels of the hierarchy, working well without information about temporal locality–information missing at all but the highest level of the hierarchy. Others use application hints to influence cache contents.

We present Karma, a global non-centralized, dynamic and informed management policy for multiple levels of cache. Karma leverages application hints to make informed allocation and replacement decisions in all cache levels, preserving exclusive caching and adjusting to changes in access patterns. We show the superiority of Karma through comparison to existing solutions including LRU, 2Q, ARC, MultiQ, LRU-SP, and Demote, demonstrating better cache performance than all other solutions and up to 85% better performance than LRU on representative workloads.

1 Introduction

Caching is used in storage systems to provide fast access to recently or frequently accessed data, with non-volatile devices used for data safety and long-term storage. Much research has focused on increasing the performance of caches as a means of improving system performance. In many storage system configurations, client and server caches form a two- or more layer hierarchy, introducing new challenges and opportunities over traditional single-level cache management. These include determining which level to cache data in and how to achieve exclusivity of data storage among the cache levels given the scant information available in all but the highest-level cache. Addressing these challenges can provide a significant improvement in overall system performance.

A *cache replacement policy* is used to decide which block is the best candidate for eviction when the cache is full. The *hit rate* is the fraction of page requests served from the cache, out of all requests issued by the application. Numerous studies have demonstrated the correlation between an increase in hit rate and application speedup [10, 12, 13, 19, 22, 27, 48, 49, 51]. This correlation motivates the ongoing search for better replacement policies. The most commonly used online replacement policy is LRU. Pure LRU has no notion of frequency, which makes the cache susceptible to pollution that results from looping or sequential access patterns [40, 47]. Various LRU variants, e.g., LRU-K [37], 2Q [25], LRFU [28], LIRS [23] and ARC [33], attempt to account for frequency as well as temporal locality.

A different approach is to manage each access pattern with the replacement policy best suited for it. This is possible, for example, by automatic classification of access patterns [13, 19, 27], or by adaptively choosing from a pool of policies according to their observed performance [4, 20]. In *informed caching*, replacement decisions are based on hints disclosed by the application itself [10, 14, 38]. Although informed caching has drawbacks for arbitrary applications (see Section 7), these drawbacks can be addressed for database systems [15, 36, 41]. File systems can also derive access patterns from various file attributes, such as the file extension or the application accessing the file. The Extensible File System [26] provides an interface which enables users to classify files and the system to derive the files' properties. Recent tools provide automatic classification of file access patterns by the file and storage systems [16]. Despite the proven advantage of informed caching, it has been employed only in the upper level cache.

The above approaches attempt to maximize the number of cache hits as a means of maximizing overall performance. However, in modern systems where both the server and the storage controller often have significant caches, a *multilevel cache hierarchy* is formed. Simply maximizing cache hits on any individual cache in a multilevel cache system will not necessarily maximize overall system performance. Therefore, given a multilevel cache hierarchy, we wish to minimize the *weighted I/O cost* which considers all data transfers between the caches and the cost of accessing each.

Multilevel cache hierarchies introduce three major problems in cache replacement. The first is the hiding of locality of reference by the upper cache [51]. The second is data redundancy, where blocks are saved in multiple cache levels [12, 35]. The third is the lack of information about the blocks' attributes (e.g., their file, the application that issued the I/O request) in the lower level caches [45].

Accesses to the low level cache are misses in the upper level. Thus, these accesses are characterized by weak temporal locality. Since LRU is based on locality of reference, its efficiency diminishes in the second level cache. Policies such as FBR [39], MultiQ [51], ARC [33] and CAR [7] attempt to solve this problem by taking into account frequency of access in addition to recency. MultiQ, for example, uses multiple LRU queues with increasing lifetimes. ARC [33] and its approximation CAR [7] distinguish between blocks that are accessed once and those that are accessed more than once. None of the above policies address the cache hierarchy as a whole, but rather manage the cache levels independently, assuming the upper level cache is managed by LRU.

In *exclusive* caching [29, 49, 51], a data block should be cached in at most one cache level at a time. One way to do this is by means of the DEMOTE operation [49]. The lower level deletes a block from its cache when it is read by the upper level. When the upper level evicts an unmodified block from its cache, the block is sent back to the lower level using DEMOTE. The lower level tries to find a place for the demoted block, evicting another block if necessary.

We propose Karma, a novel approach to the management of multilevel cache systems which attempts to address all of the above issues in concert. Karma manages all levels in synergy. We achieve exclusiveness by using application hints at all levels to classify all cached blocks into disjoint sets and partition the cache according to this classification. We distinguish between a READ, which deletes the read block from a lower level cache, and a READ-SAVE, which instructs a lower level to save a block in its cache (this distinction can be realized using the existing SCSI command set, by setting or clearing the *disable page out* (DPO) bit in the READ command [1]). We also use DEMOTE to maintain exclusiveness in partitions that span multiple caches. By combining these mechanisms, Karma optimizes its cache content according to the different access patterns, adjusting to patterns which change over time.

The hints divide the disk blocks into sets based on their expected access pattern and access frequency. Each set is allocated its own cache partition, whose size depends on the frequency of access to the set, its size, and the access pattern of its blocks. Partitions accessed with higher frequency are placed at a higher cache level. Each partition is managed by the replacement policy most suited for its set. Since the lower cache levels are supplied with the application's hints, they can independently compute the partitioning, allocating space only for partitions that do not fit in the upper levels.

Karma is applicable to any application which is able to provide general hints about its access patterns. Databases are a classic example of such applications, where the access pattern is decided in advance by the query optimizer. We base our experimental evaluation both on real database traces, and on synthetic traces with Zipf distribution. For the real traces, we used the *explain* mechanism of the PostgreSQL database as a source of application hints. For the synthetic workload, we supplied Karma with the access frequency of the blocks in the data set. We simulated a hierarchy of two cache levels and one storage level for comparing Karma to LRU, 2Q [25], ARC [33], MultiQ [51], LRU-SP [10] and Demote [49]. We also defined and implemented extensions to these policies to apply to multiple levels of cache. The comparison is by means of the weighted I/O cost.

Karma compares favorably with all other policies: its use of application hints enables matching the optimal policy to each access pattern, its dynamic repartitioning eliminates the sensitivity to changing access patterns and its exclusive caching enables exploitation of every increase in the aggregate cache size. When the aggregate cache size is very small (less than 3% of the data set), Karma suffers from the overhead of DEMOTE, as demoted blocks are discarded before being re-accessed. For all other cache sizes Karma shows great improvement over all other policies. Karma improves over LRU's weighted I/O cost by as much as 85% on average on traces which are a permutation of queries. On such traces, Karma shows an additional 50% average improvement over the best LRU-based policy (Demote) when compared to LRU and it shows an additional 25% average improvement over the best informed policy (LRU-SP).

The rest of the paper is organized as follows. Our model is defined in Section 2. In Section 3 we define marginal gains, and in Section 4 we describe Karma's policy. The experimental testbed is described in Section 5, with our results in Section 6. Section 7 describes related work. We conclude in Section 8.

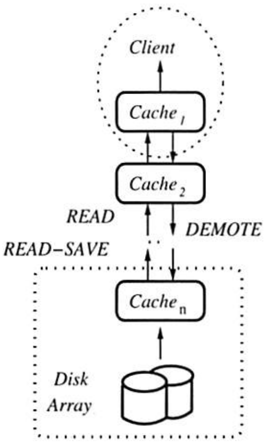


Figure 1: Our storage model consists of n levels of cache, with one cache in each level. The operations are READ and READ-SAVE from $Cache_i$ to $Cache_{i-1}$, and DEMOTE from $Cache_{i-1}$ to $Cache_i$.

2 Model Definition

As shown in Figure 1, our model consists of one client, n cache levels, $Cache_1$,...,$Cache_n$, and one storage level, $Disk$, arranged in a linear hierarchy. we refer to more complicated hierarchies in Section 8. $Cache_i$ is of size S_i, and access cost C_i. The cost of a disk access is C_{Disk}. The cost of demoting a block from $Cache_{i-1}$ to $Cache_i$ is D_i. We assume that a lower cache level carries an increased access cost, and that demoting and access costs are equal for a given level. Namely,
$C_1 = D_1 < C_2 = D_2 < \ldots < C_n = D_n < C_{Disk}$.

Typically, $Cache_1$ resides in the client's memory and $Cache_n$ resides on the storage controller. Additional cache levels may reside in either of these locations, as well as additional locations in the network. The access costs, C_i and D_i, represent a combination of computation, network, and queuing delays. C_{Disk} also includes seek times.

The model is demand paging, read-only (for the purpose of this work, we assume a separately managed write cache [18]), and defines three operations:

- READ (x, i)—move block x from $cache_{i+1}$ to $cache_i$, removing it from $cache_{i+1}$. If x is not found in $cache_{i+1}$, READ$(x, i+1)$ is performed recursively, stopping at $Disk$ if the block is not found earlier.
- READ-SAVE (x, i)—copy block x from $cache_{i+1}$ to $cache_i$. Keep block x in $cache_{i+1}$ only if its range is allocated space in $cache_{i+1}$. If x is not in $cache_{i+1}$, READ-SAVE$(x, i+1)$ is performed recursively, stopping at the $Disk$ if the block is not found earlier.
- DEMOTE (x, i)—move block x from $cache_i$ to $cache_{i+1}$, removing it from $cache_i$.

The *weighted I/O cost* of a policy on a trace is the sum of costs of all READ, READ-SAVE and DEMOTE operations it performs on that trace.

3 Marginal Gain

The optimal offline replacement policy for a single cache is Belady's MIN [9]. Whenever a block needs to be evicted from the cache, MIN evicts the one with the largest *forward distance* – the number of distinct blocks that will be accessed before this block is accessed again. To develop our online multilevel algorithm, we have opted to use application hints in a way which best approximates this forward distance. To this end, we use the notion of *marginal gains*, which was defined in previous work [36].

The marginal gain for an access trace is the increase in hit rate that will be seen by this trace if the cache size increases by a single block:
$MG(m) = Hit(m) - Hit(m-1)$,
where $Hit(m)$ is the expected hit rate for a cache of size m. Below we show how $MG(m)$ is computed for three common access patterns: looping, sequential, and random. Although we focus on these three patterns, similar considerations can be used to compute the marginal gain of any other access pattern for which the hit rate can be estimated [14, 27, 37, 38].

Obviously, the marginal gain depends on the replacement policy of the cache. We assume that the best replacement policy is used for each access pattern: MRU (Most Recently Used) is known to be optimal for sequential and looping references, whereas LRU is usually employed for random references (for which all policies perform similarly [10, 13, 14, 15, 27, 41]).

Sequential accesses. For any cache size m, since no block is previously referenced, the hit rate for a sequential access trace is $Hit_{seq}(m) = 0$. Thus, the resulting marginal gain is 0 as well.

Random (uniform) accesses. For an access trace of R blocks of uniform distribution, the probability of accessing each block is $1/R$ [36]. For any cache size $m \leq R$, the hit rate is thus $Hit_{rand}(m) = m/R$. The resulting marginal gain is:

$$MG_{rand}(m) = \begin{cases} m/R - (m-1)/R = 1/R & m \leq R \\ 0 & m > R. \end{cases}$$

Looping accesses. The *loop length* of a looping reference is the number of blocks being re-referenced [27]. For a looping reference with loop length L, the expected hit rate for a cache of size m managed by MRU is $Hit_{loop}(m) = min(L, m)/L$. Thus,

$$MG_{loop}(m) = \begin{cases} m/L - (m-1)/L = 1/L & m \leq L \\ L/L - L/L = 0 & m > L. \end{cases}$$

In other words, the marginal gain is constant up to the point where the entire loop fits in the cache and from there on, the marginal gain is zero.

We deal with traces where accesses to blocks of several ranges are interleaved, possibly with different access patterns. Each range of blocks is accessed with one pattern. In order to compare the marginal gain of references to different ranges, we use the frequency of access to each range. Let F_i be the percent of all accesses which address range i. Define the *normalized expected hit rate* for range i as $Hit_i(m) \times F_i$, and the *normalized marginal gain* for range i as $NMG_i(m) = MG_i(m) \times F_i$.

Although marginal gains are defined for a single level cache and measure hit rate rather than weighted I/O cost, normalized marginal gains induce an order of priority on all ranges – and thus on all blocks – in a trace. This order is used by our online management algorithm, Karma, to arrange the blocks in a multilevel cache system: the higher the range priority, the higher its blocks are placed in the cache hierarchy. This strategy maximizes the total normalized marginal gain of all blocks stored in the cache.

Note that when all blocks in a range have the same access frequency there is a correlation between the normalized marginal gain of a range and the probability that a block of this range will be accessed. A higher marginal gain indicates a higher probability. Therefore, there is no benefit in keeping blocks of ranges with low marginal gains in the cache: the probability that they will be accessed is small. Ranges with several access frequencies should be divided into smaller ranges according to those frequencies (see the example in Figure 2, described in Section 4).

A major advantage of basing caching decisions on marginal gains is the low level of detail required for their computation. Since only the general access pattern and access frequency are required, it is much more likely that an application be able to supply such information. Our experience shows that databases can supply this information with a high degree of accuracy. We expect our hints can also be derived from information available to the file system [16, 26].

4 Karma

Karma calculates the normalized marginal gain of a range of blocks (which corresponds to each of the sets described in Section 1) of blocks and then uses it to indicate the likelihood that the range's blocks will be accessed in the near future. To calculate the normalized marginal gain, Karma requires that all accessed disk blocks be classified into ranges. The following information must be provided (by means of application hints) for each range: an identifier for the range, its access pattern, the number of blocks in the range, and the frequency of access to this range. The computation is described in Section 3. Each block access is tagged with the block's range identifier, enabling all cache levels to handle the block according to its range.

Karma allocates for each range a fixed cache partition in a way that maximizes the normalized marginal gain of the blocks in all cache levels. It places ranges with higher normalized marginal gain in higher cache levels, where the access cost is lower. More precisely: space is allocated in $Cache_i$ for the ranges with the highest normalized marginal gain that were not allocated space in any $Cache_j$, $j < i$. For each level i there can be at most one range which is split and is allocated space in both level i and the adjacent lower level $i + 1$. Figure 2 shows an example of Karma's allocation.

Each range is managed separately, with the replacement policy best suited for its access pattern. When a block is brought into the cache, a block from the same range is discarded, according to the range's policy. This maintains the fixed allocation assigned to each range.

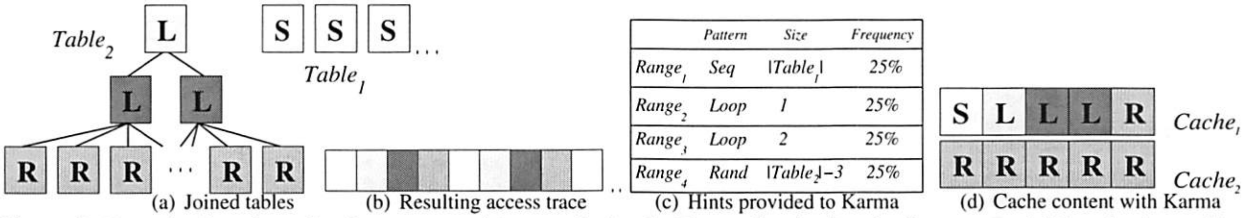
The amount of cache space required for maintaining the information about the ranges and the data structures for the cache partitions is less than one cache block. The pseudocode for Karma appears in Figure 3.

Hints. Karma strongly depends on the ability to propagate application information to the lower cache levels. Specifically, the range identifier attached to each block access is crucial for associating the block with the knowledge about its range. A method for passing information (sufficient for Karma) from the file system to the I/O system was suggested [8] and implemented in a Linux 2.4.2 kernel prototype.

For the two tables joined in the example in Figure 2, Karma will be supplied with the partitioning of the blocks into tables and index tree levels, as in Figure 2(c). Additionally, each cache level must know the aggregate size of all cache levels above it. Such information can be passed out-of-band, without changing current I/O interfaces. Each block access will be tagged with its range identifier, enabling all cache levels to classify it into the correct partition.

As in all informed management policies, Karma's performance depends on the quality of the hints. However, thanks to its exclusive caching, even with imperfect hints Karma will likely outperform basic LRU at each level. For example, with no hints provided and the entire data set managed as one partition with LRU replacement, Karma essentially behaves like Demote [49].

Allocation. Allocating cache space to blocks according to their normalized marginal gain would result in zero allocation for sequential accesses. Yet, in such patterns the application often accesses one block repeatedly before moving on to the next block. In some database queries, for example, a block may be accessed a few times, until all tuples in it have been processed. Therefore, ranges accessed sequentially are each allocated a



	Pattern	Size	Frequency
$Range_1$	Seq	$\lvert Table_1\rvert$	25%
$Range_2$	Loop	1	25%
$Range_3$	Loop	2	25%
$Range_4$	Rand	$\lvert Table_2\rvert-3$	25%

(a) Joined tables (b) Resulting access trace (c) Hints provided to Karma (d) Cache content with Karma

Figure 2: Karma's allocation of buffers to ranges in two cache levels. The application is a database query. (a) Two database tables are joined by scanning one of them sequentially and accessing the second one via an index. (b) The resulting access pattern is an interleaving of four ranges: one sequential (S), two loops (L), and one random (R). (c) This partitioning into ranges is supplied to Karma at the beginning of query execution. (d) Karma allocates one buffer for the sequential accesses (see text), three to hold the root and inner nodes of the index, and the remaining space to the randomly accessed blocks.

single block in the upper level cache (if prefetching is added to the model, several blocks may be allocated to allow for sequential read-ahead).

We allow for locality within the currently accessed block by always bringing it into $Cache_1$. When this block belongs to a range with no allocated space in $Cache_1$, we avoid unnecessary overhead by reading it using READ-SAVE, and discarding it without using DEMOTE.

Lazy repartitioning. When Karma is supplied with a new set of hints which indicate that the access patterns are about to change (e.g., when a new query is about to be processed), it repartitions all cache levels. Cache blocks in each level are assigned to the new partitions. As a result, disk blocks which do not belong to any of the new partitions in the cache level where they are stored become candidates for immediate eviction.

A block is rarely assigned a new range. For example, in a database blocks are divided according to their table or index level. Therefore, when a new query is processed the division into ranges will remain the same; only the access frequency and possibly the access pattern of the ranges will change. As a result, the blocks will remain in the same partition, and only the space allocated for this partition will change. This means that the sizes of the partitions will have to be adjusted, as described below.

Karma never discards blocks while the cache is not full, nor when the cache contains blocks which are immediate candidates for eviction (blocks from old ranges or blocks that were READ by an upper level). Karma ensures that even in transitional phases (between the time a new set of hints is supplied and the time when the cache content matches the new partitioning) the cache keeps the blocks with the highest marginal gain. As long as a cache (at any level) is not full, non-sequential ranges are allowed to exceed the size allocated for them. When no space is left and blocks from ranges with higher marginal gain are accessed, blocks from ranges which exceeded their allocated space are first candidates for eviction, in reverse order of their marginal gain.

Note that during repartitioning the cache blocks are not actually moved in the cache, as the partitioning is logical, not physical. The importance of handling this transitional stage is that it makes Karma less vulnerable to the order in which, for example, several queries are processed in a row.

Replacement. Karma achieves exclusive caching by partitioning the cache. This partitioning is maintained by use of DEMOTE and READ-SAVE, where each cache level stores only blocks belonging to its assigned ranges. For each i, $1 \leq i \leq n-1$, $Cache_i$ demotes all evicted blocks which do not belong to sequential ranges. When $Cache_i$ is about to read a block without storing it for future use, it uses READ-SAVE in order to prevent $Cache_{i+1}$ from discarding the block. Only one such block is duplicated between every two cache levels at any moment (see Figure 3). For each j, $2 \leq j \leq n$, $Cache_j$ does not store any READ blocks, but only those demoted by $Cache_{j-1}$ or read using READ-SAVE.

Special attention must be given to replacement in ranges which are split between adjacent cache levels i and $i+1$. The LRU (or MRU) stack must be preserved across cache levels. $Cache_i$ manages the blocks in a split range with the corresponding policy, demoting all discarded blocks. $Cache_{i+1}$ inserts demoted blocks at the most recently used (MRU) position in the stack and removes them from the MRU or LRU position, according to the range's policy. Blocks READ by $Cache_i$ are removed from the stack and become immediate candidates for eviction. This way, the stack in $Cache_{i+1}$ acts as an extension of the stack in $Cache_i$.

5 Experimental Testbed

While Karma is designed to work in n levels of cache, in our experiments we compare it to existing algorithms on a testbed consisting of two cache levels ($n = 2$).

5.1 PostgreSQL Database

We chose PostgreSQL [34] as a source of application hints. Each of its data files (table or index) is divided into disk blocks. Accesses to these blocks were traced

```
Partition (Block $X$)
  Return the partition to which $X$ belongs

Transition (Block $X$)
  Return (cache is not full) or (LowestPriority($X$) $<$ Partition($X$))

LowestPriority(Block $X$)
  Return partition with lowest priority exceeding its allocated size.
  If none exists return Partition($X$).

Evict(Block $X$)
  If ($X$ was in Reserved) or (Partition($X$) is Seq)
    Discard $X$
  Else
    DEMOTE $X$
    Discard $X$

Insert (Block $X$, Partition $P$)
  If (cache is not full)
    Put $X$ in $P$
  Else
    Remove block $Z$ from LowestPriority($X$)
  Evict ($Z$)
  Put $X$ in $P$

Hit (Block $X$, Action $A$)
  If (first level) or (A = DEMOTE/READ-SAVE)
    update place in stack
  Else // low level and A = READ
    Put $X$ in ReadBlocks
```

```
Miss (Block $X$, Action $A$)
  If (low level) and (A = READ)
    READ $X$
    Discard $X$
  Else // (first level) or (A = DEMOTE/READ-SAVE)
    If (Partition($X$) fits in the cache) or (cache is not full)
      If ($A \neq$ DEMOTE)
        READ $X$
      Insert ($X$,Partition($X$))
    Else // Partition($X$) doesn't fit at all
      If (first level) // (A $\neq$ DEMOTE)
        READ-SAVE $X$
        Remove block $Y$ from Reserved
        If (Transition($Y$))
          Insert ($Y$, Partition($Y$))
        Else
          Discard $Y$
        Put $X$ in Reserved
      Else // low level
        If (Transition($X$))
          If (A $\neq$ DEMOTE)
            READ X
          Insert ($X$,Partition($X$))
        Else // no space for $X$
          If ($A \neq$ DEMOTE)
            READ-SAVE $X$
          Discard $X$
```

Figure 3: Pseudocode for Karma. ***Reserved***—A reserved buffer of size 1 in the first cache level for blocks belonging to ranges with low priority. ***ReadBlocks***—A low priority partition holding blocks that were READ by an upper level cache and are candidates for eviction. ***READ, READ-SAVE, DEMOTE***—The operations defined by the model in Section 2.

by adding trace lines to the existing debugging mechanism.

Like most database implementations, PostgreSQL includes an *explain* mechanism, revealing to the user the plan being executed for an SQL query. This execution plan determines the pattern and frequency with which each table or index file will be accessed during query execution. We used the output of explain to supply Karma with the characterization of the access pattern for each range, along with a division of all the blocks into ranges. Blocks are first divided by their table, and in some cases, a table can be sub-divided into several ranges. For example, in B-tree indices each level of the tree is characterized by different parameters (as in Figure 2).

5.2 TPC Benchmark H Traces

The TPC Benchmark H (TPC-H) is a decision support benchmark defined by the Transaction Processing Council. It exemplifies systems that examine large volumes of data and execute queries with a high degree of complexity [2]. In our experiments we used the default implementation provided in the benchmark specification to generate both the raw data and the queries.

We simulated our cache replacement policies on two types of traces:

- Repeated queries: each query is repeated several times, requesting different values in each run. This trace type models applications that access the database repeatedly for the same purpose, requesting different data in different iterations.
- Query sets: each query set is a permutation of the 22 queries of the benchmark, executed serially [2]. The permutations are defined in Appendix A of the benchmark specification. The query sets represent applications that collect various types of data from the database.

Queries 17, 19, 20, 21 and 22 have especially long execution traces (each over 25,000,000 I/Os). As these traces consist of dozens of loop repetitions, we used for our simulation only the first 2,000,000 I/Os of each trace.

5.3 Synthetic Zipf Workload

In traces with Zipf distribution, the frequency of access to block i is proportional to $1/i^{\alpha}$, for α close to 1. Such distribution approximates common access patterns, such as file references in Web servers. Following previous studies [37, 49], we chose Zipf as a non-trivial random workload, where each block is accessed at a different, yet predictable frequency. We use settings similar to those used in previous work [49] and set $\alpha = 1$, for 25,000 different blocks.

Karma does not require information about the access frequency of each block. We supplied it with a parti-

Alg	Basic-Alg	Double-Alg	Global-Alg
LRU	Basic-LRU (LRU+LRU)	Double-LRU	Demote
2Q	Basic-2Q (LRU+2Q)	**Double-2Q**	**Global-2Q**
ARC	Basic-ARC (LRU+ARC)	**Double-ARC**	**Global-ARC**
MultiQ	Basic-MultiQ (LRU+MultiQ)	**Double-MultiQ**	**Global-MultiQ**
LRU-SP	Basic-LRU-SP (LRU-SP+LRU)	N/A	***(SP-Karma)***

Table 1: The policies in our comparative framework. In addition to the policies known from the literature, we defined the Double and Global extensions indicated in bold. We compared Karma to all applicable Basic and Double policies, as well as to Demote and to Global-MultiQ.

tioning of the blocks into several ranges and the access frequency of each range. This frequency can be easily computed when the distribution parameters are available.

Although the blocks in each range have different access frequencies, for any two blocks i and j, if $i < j$ then the frequency of access to block i is greater then that of block j. The blocks are assigned to ranges in increasing order, and so for any two ranges I and J, if $I < J$ then all the blocks in range I have greater access frequencies than the blocks in range J.

5.4 Comparative framework

We compared Karma to five existing replacement algorithms: LRU, 2Q, ARC, MultiQ, and LRU-SP, each of which we examined using three different approaches: *Basic*, *Double*, and *Global*. In the Basic approach, each algorithm is used in conjunction with LRU, where the existing algorithm is used on one level and LRU on the other, as was defined in the respective literature. In the Double approach, each is used on both cache levels. The third approach is a global management policy, where each algorithm must be explicitly adapted to use DEMOTE.

Although Global-2Q and Global-ARC were not actually implemented in our experiments, we describe, in what follows, how they would work in a multilevel cache. It is also important to note that Global-MultiQ is not an existing policy: we defined this algorithm for the purpose of extending the discussion, and it is implemented here for the first time. We refer to the special case of LRU-SP in Section 5.5. The algorithms and approaches are summarized in Table 1. The actual experimental results are described in Section 6.

Least Recently Used (LRU). LRU is the most commonly used cache management policy. *Basic-LRU* and *Double-LRU* are equivalent, using LRU on both cache levels. *Global-LRU* is the Demote policy [49], where the upper level cache demotes all blocks it discards. The lower level cache puts blocks it has sent to the upper level at the head (closest to being discarded end) of its LRU queue, and puts demoted blocks at the tail.

2Q. 2Q [25] uses three queues. One LRU queue, A_m, holds "hot" pages that are likely to be re-referenced. A second FIFO queue, A_{in}, holds "cold" pages that are seen only once. The third LRU queue, A_{out}, is a ghost cache, holding meta-data of blocks recently evicted from the cache. As 2Q was originally designed for a second level cache, *Basic-2Q* uses LRU at the first cache level and 2Q at the second. *Double-2Q* uses 2Q at both cache levels. *Global-2Q* keeps A_{out} and A_{in} in the second level cache, dividing A_m between both cache levels. In all these cases, we use the optimal parameters for each cache level [25]. A_{in} holds 25% of the cache size and A_{out} holds blocks that would fit in 50% of the cache.

ARC. ARC [33] was also designed for a second level cache. It divides blocks between two LRU queues, L_1 and L_2. L_1 holds blocks requested exactly once. L_2 holds blocks requested more than once. The bottom (LRU) part of each queue is a ghost cache. The percentage of cache space allocated to each queue is dynamically adjusted, and history is saved for as many blocks that would fit in twice the cache size. *Basic-ARC* uses LRU at the first cache level and ARC at the second. *Double-ARC* uses ARC at both cache levels. *Global-ARC* keeps the ghost caches in the second level cache, as well as the LRU part of what is left of L_1 and L_2. ARC is implemented for each cache level with dynamic adjustment of the queue sizes [33].

MultiQ. MultiQ [51] was originally designed for a second level cache. It uses multiple LRU queues, each having a longer lifetime than the previous one. When a block in a queue is accessed frequently, it is promoted to the next higher queue. On a cache miss, the head of the lowest non-empty queue is evicted. *Basic-MultiQ* uses LRU at the first cache level and MultiQ at the second. *Double-MultiQ* uses MultiQ at both cache levels. We implemented MultiQ for each cache level with 8 queues and a ghost cache. The *Lifetime* parameter is set according to the observed temporal distance. We extended MultiQ to *Global-MultiQ* in a straightforward way. The ghost cache is allocated in the second level cache, and the queues are divided dynamically between the cache levels, with at most one queue split between the levels. Whenever a block is brought into the first level cache, the block at the bottom of the lowest queue in this level is demoted to the second level cache.

5.5 Application Controlled File Caching

In **LRU-SP** [10], applications may use specific interface functions to assign priorities to files (or ranges in files). They may also specify cache replacement policies for each priority level. Blocks with the lowest priority are first candidates for eviction. In the original paper, applications were modified to include calls to these interface functions.

As a policy assuming hints from the application, LRU-SP is designed for a first level cache. Therefore, we implement *Basic-LRU-SP* with LRU at the second level. *Double-LRU-SP* would let the application manage each cache level directly yet independently. This seems unreasonable and thus we did not implement this extension. Without hints available at the second level cache, the simple addition of DEMOTE will not result in global management, since informed decisions cannot be made in the second level. Thus, extending LRU-SP to *Global-LRU-SP* is not applicable.

A new multilevel extension. To evaluate the contribution of the different mechanisms of Karma, we defined a new policy, *SP-Karma*, for managing multilevel caches, which added to LRU-SP most of the features we defined in Karma. This extension resulted in a new cache management algorithm which is similar to Karma and allows us to better understand the value of specific attributes of Karma. In particular, we added DEMOTE for cooperation between cache levels, we derived priorities using Karma's calculation of normalized marginal gains (this mechanism was also used to supply hints to Basic-LRU-SP above), and we supplied these priorities to both cache levels. Since SP-Karma now bases its decisions on Karma's priorities, the significant difference between our two policies is the use of READ-SAVE.

The results of SP-Karma resembled those of Karma, but Karma achieved better performance on all traces. The use of READ-SAVE resulted in Karma executing fewer DEMOTE operations, thus outperforming SP-Karma by up to 1% in the large caches and 5% in the small ones. Since we defined our new policy, SP-Karma, to be very similar to Karma and since it shows similar results, in Section 6 we compare only Karma to the existing policies.

6 Results

We simulated the policies described in Section 5.4 on a series of increasing cache sizes and measured the weighted I/O cost:

$$\begin{aligned} \textit{Weighted I/O cost} = \; & C_2 \times (\text{misses in } Cache_1) \\ & + D_2 \times (\text{DEMOTEs to } Cache_2) \\ & + C_{Disk} \times (\text{misses in } Cache_2) \end{aligned}$$

For all experiments (excluding the one depicted in Figure 9), we set $C_2 = D_2 = 1$ and $C_{Disk} = 20$, as in [49]. For all experiments, excluding the one depicted in Figure 10, we assume, as in previous studies [11, 49], that the caches are of equal size ($S_1 = S_2$). The cache size in the graphs refers to the aggregate cache size as a fraction of the size of the data set. Thus, a cache size of 1 means that each cache level is big enough to hold one-half of the data set. A cache size of 2 means that the entire data set fits in the top level, and all policies should perform equally.

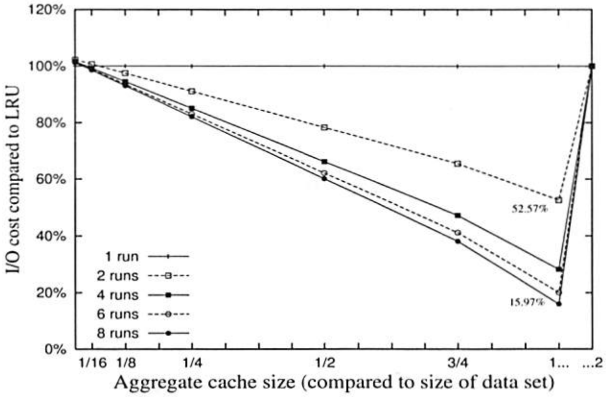


Figure 4: Karma's improvement over LRU on Query 3 run multiple times.

The results are normalized to their weighted I/O cost compared to that incurred by LRU. This gives a better representation of the improvement over the default LRU management, making it easier to compare the policies.

How does Karma compare to Basic-LRU? We ran all policies on traces of each query, using several cache sizes. Karma generally yielded similar curves for all queries. In Figure 4 we take a close look at Query 3 as a representative example, in order to understand the behavior of Karma with different cache sizes. Query 3 is sequential, scanning 3 tables. PostgreSQL creates a hash table for each database table and joins them in memory. Subsequent runs of Query 3 result in a looping reference.

We ran Query 3 eight times, to show how the I/O cost incurred by Karma decreases, in comparison to that of LRU, as the number of runs increases. LRU suffers from the three problems of multilevel caching. The second level cache experiences no locality of reference, as all repeated accesses are hidden by the first level. Even when the aggregate cache is as large as the data set LRU does not exploit the entire space available due to redundancy between levels. Finally, when the cache is smaller than the data set LRU is unable to maintain used blocks until the next time they are accessed, and so it does not benefit from increasing the cache until its size reaches the size of the data set. Karma does not suffer any of those drawbacks, and its weighted I/O cost decreases significantly as the cache size increases. Although the portion of the loop which fits in the cache is equal for all runs (for each cache size), the hit rate of LRU remains zero, while Karma's hit rate increases in both cache lev-

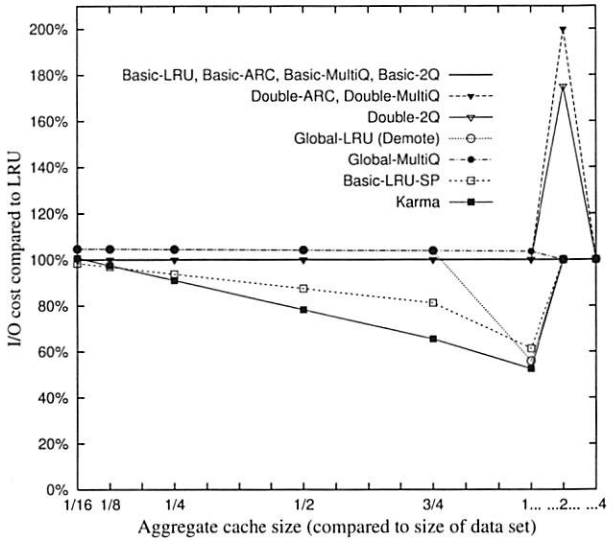


Figure 5: Weighted I/O cost for all policies on Query 3 run twice. Policies with identical results are plotted using a single line. Other than Karma, only the Global policies are able to fit the entire data set in the cache when the aggregate cache size equals the size of the data set. Unlike the non-informed policies, Karma and Basic-LRU-SP reduce their I/O cost gradually as the cache size increases.

els. This causes an increasing improvement in Karma's weighted I/O cost compared to that of LRU.

How does Karma compare to single level non-informed policies? Figure 5 contains the results for all policies on Query 3. The Basic implementations of 2Q, MultiQ, and ARC behave like LRU. Their division into separate LRU queues does not help in identifying the looping reference if it is larger than the cache size. When the cache size increases to the point where the entire loop fits in the cache, there can be no improvement over LRU.

The Double extensions of 2Q, MultiQ, and ARC allocate a ghost cache (whose size is 0.22%, 0.16% and 0.43% of the cache size, respectively) in both cache levels. This small portion is sufficient to prevent the data set from fitting in one level when the cache size is 2, leading to worse performance than LRU.

Karma is informed of the looping reference and manages it with MRU replacement. This enables Karma to benefit from every increase in the cache size. When the entire loop fits in the aggregate cache Karma benefits from its exclusive caching and shows the largest improvement over the I/O cost of LRU. We refer to the other global and informed policies later in this section.

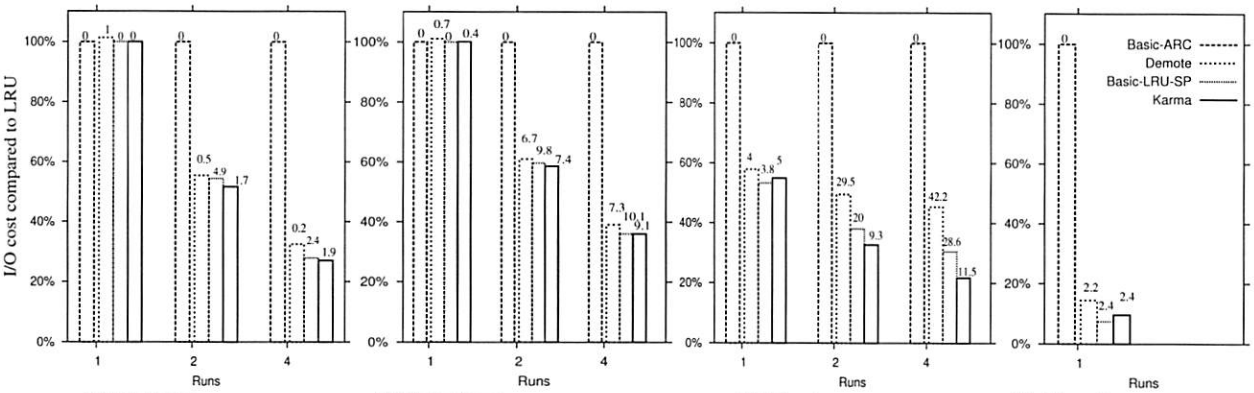
The results for the other queries are similar to those for Query 3. Figure 6 summarizes the results for multiple runs of each query, comparing the weighted I/O cost of Karma to three representative policies. Each query was executed four times, with different (randomly generated) parameters. The traces are a concatenation of one to four of those runs, and the results are for an aggregate cache size which can contain the entire data set. Most of the queries were primarily sequential, causing all policies to perform like LRU for a single run. Repeated runs, however, convert the sequential access into a looping access, which is handled poorly by LRU when the loop is larger than the cache. These experiments demonstrate the advantage of policies which are not based purely on recency.

The queries were of four basic types, and the results are averaged for each query type. The first type includes queries which consist only of sequential table scans. Karma's I/O cost was lower than that of LRU and LRU-based policies on these queries by an average of 73% on four runs (Figure 6(a)). In the second type, an index is used for a table scan. Indices in PostgreSQL are constructed as B-trees. An index scan results in some locality of reference, improving LRU's performance on query traces of this type, compared to its performance on queries with no index scans. Karma's I/O cost on these queries was lower than that of LRU by an average of 64% on four runs (Figure 6(b)). In the third query type, one or more tables are scanned several times, resulting in a looping pattern within a single execution of the query. In queries of this type, I/O cost lower than that of LRU is obtained by some policies even for one run of the query. Four runs of each of these queries consisted of eight runs of the loop, resulting in significant reduction in I/O cost as compared to the other query types. Karma's I/O cost on these queries was lower than that of LRU by an average of 78.33% on four runs (Figure 6(c)). In the last type, each query instance included dozens of iterations over a single table. These traces were trimmed to the first 2,000,000 I/Os, which still contained more than ten loop iterations. On a single run of these queries Karma performed better than LRU by an average of 90.25% (Figure 6(d)).

The results in Figure 6 for Basic-ARC correspond to those in Figure 5. Like the rest of the LRU-based policies constructed for a single cache level, it is not exclusive, and thus it is unable to exploit the increase in cache size and fit the data set in the aggregate cache.

To see how the different policies perform on more heterogeneous traces, we use the query sets described in Section 5.2, which are permutations of all queries in the benchmark. Our results for the first 20 sets are shown in Figure 7 for the largest cache sizes, where the policies showed the most improvement. The left- and right-hand columns show results for an aggregate cache size that can hold half of the data set or all of it, respectively.

The I/O cost of Basic-ARC and Double-ARC is not much lower than that of LRU. ARC is designed to handle traces where most of the accesses are random or exhibit locality of reference. Its dynamic adjustment is aimed at



(a) No Index (1,3,6,10,14) (b) Index Scan (2,4,5,7,8,9,12,13,16,21) (c) Two Loops (11,15,18) (d) Many Loops (17,19,20,22)

Figure 6: Improvement in I/O cost of Karma, Basic-ARC, Demote and Basic-LRU-SP compared to LRU, on repeated runs of all queries. The columns show the average I/O cost of these policies for each query type, when the aggregate cache size equals the size of the data set. Each column is tagged with the standard deviation for this query type. Karma improved over the I/O cost of LRU on repeated runs of the queries by an average of 73%, 64%, 78% and 90% on query types a, b, c, and d, respectively.

preventing looping and sequential references from polluting the cache. It is not designed to handle traces where larger segments of the trace are looping or sequential.

The I/O cost of Basic-2Q and Basic-MultiQ is lower than that of LRU for the larger aggregate cache. Both policies benefit from dividing blocks into multiple queues according to access frequency. Double-2Q outperforms Basic-2Q by extending this division to both cache levels. Double-MultiQ, however, suffers when the ghost cache is increased in two cache levels and does not show average improvement over Basic-MultiQ. The high standard deviation of the Basic and Double extensions of 2Q and MultiQ demonstrate their sensitivity to the order of the queries in a query set, and consequently, their poor handling of transitional stages.

Karma outperforms all these policies. Its reduction in I/O cost is significantly better than that of the non-informed single level policies. This reduction is evident not only when the entire data set fits in the cache, but in smaller cache sizes as well. The low standard deviation shows that it copes well with changing access patterns resulting in transitional stages.

Figure 8 shows how Karma compares to existing policies on a Zipf workload. We chose to present the results for Double-ARC because it showed better performance than all non-informed policies that were designed for a single cache level. This is because ARC avoids caching of blocks that are accessed only once in a short period of time. In a Zipf workload, these are also the blocks which are least likely to be accessed again in the near future. Note that ARC's improvement is most evident when the cache is small. When the cache is larger, such blocks occupy only a small portion of the cache in LRU, and so the benefit of ARC is less distinct.

Karma improves over the I/O cost of LRU by as much as 41%, adding as much as 25% to the improvement of

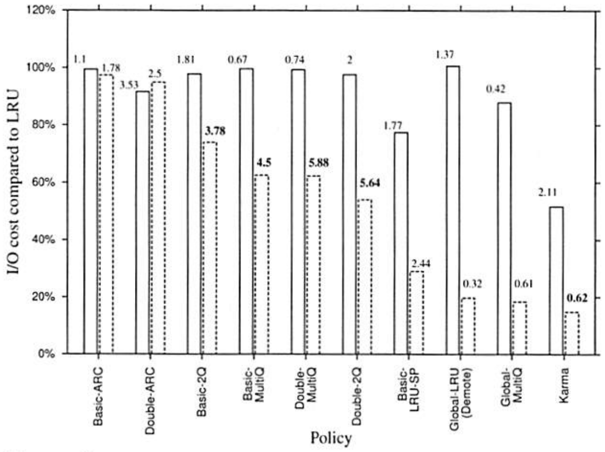


Figure 7: Weighted I/O cost of Karma and the existing policies on 20 query sets, as compared to LRU. The columns show the weighted I/O cost for each policy averaged over 20 query sets for each of two cache sizes. The left bar for each policy is for an aggregate cache size of 1/2 and the right one is for 1. Each column is tagged with the standard deviation for this policy.The policies are sorted in descending order of their I/O cost for the large cache size. Karma shows improvement over all cache sizes by combining knowledge of the access pattern with exclusive caching. It improves the I/O cost of LRU by 47% and 85% for the small and big cache sizes, respectively.

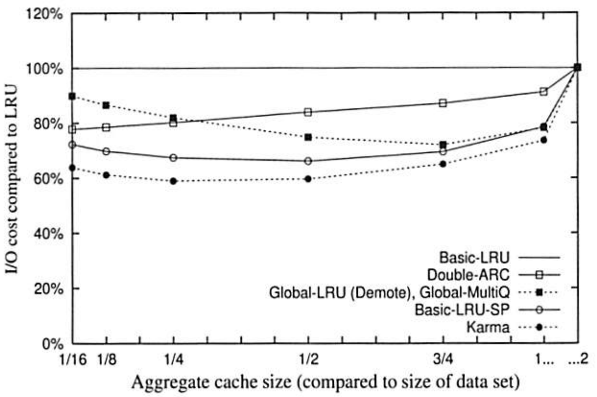


Figure 8: Weighted I/O cost for selected policies on a Zipf workload. Even when the access pattern exhibits strong temporal locality, Karma outperforms all LRU-based policies and the basic hint-based policy. Karma improves over the I/O cost of LRU by at least 26%, and by as much as 41%.

Double-ARC. Since the access frequency of the blocks is available to Karma, it does not bring into the cache blocks with low frequency. Exclusiveness further increases its effective cache size, resulting in improved performance.

How does Karma compare to global non-informed policies? When run on Query 3, Demote (Global-LRU) (Figure 5) exhibits different behavior than the single level policies. It manages to reduce the I/O cost significantly when the entire data set fits in the aggregate cache. This is the result of the exclusive caching achieved by using DEMOTE. Still, for smaller cache sizes, it is not able to "catch" the looping reference and does not benefit from an increase in cache size. Instead, its overhead from futile DEMOTE operations means that its performance is worse than LRU's for all cache sizes in which the aggregate cache cannot contain all the blocks. We expected Global-MultiQ to perform at least as well as Demote, but due to its large ghost cache it is unable to fit the entire data set into the aggregate cache, even when the cache is large enough. Instead, it only suffers the overhead of futile DEMOTE operations. Being a global policy, Karma is able to exploit the entire aggregate cache. Since it manages the looping accesses with MRU replacement, it improves gradually as the cache size increases.

Figure 6 shows only the results for the largest cache size, where Demote shows its best improvement. In fact, we expect any global policy to achieve this improvement when the entire data set fits in the aggregate cache. Even there, Karma achieves lower I/O cost than Demote, thanks to its use of READ-SAVE and MRU management for loops. Unlike Demote, Karma achieves this reduction in smaller cache sizes as well.

The performance of the global policies on the query sets is represented well in Figure 7. It is clear that when the cache size is smaller than the data set, they are not able to improve the I/O cost of LRU significantly. This improvement is only achieved when the entire data set fits in the aggregate cache (represented by the right-hand column). Karma, however, improves gradually as the cache size increases. Combining exclusive management with application hints enables it to maximize the cache performance in all cache sizes.

When the access pattern does not include looping references (Figure 8), the global policies improve gradually as the cache size increases. Although a Zipf workload exhibits significant locality of reference, adding exclusiveness to LRU does not achieve good enough results. In the smallest cache size Demote and Global-MultiQ improve over the I/O cost of LRU by 10%. Karma, with its knowledge of access frequencies, achieves additional improvement of 26%.

How does Karma compare to hint-based policies? Basic-LRU-SP using Karma's hints and priorities performs better than any non-informed single level policy, for all our traces. Using hints enables it to optimize its use of the upper level cache. When the access pattern is looping, the combination of MRU management in the upper cache and default LRU in the lower results in exclusive caching without the use of DEMOTE. Note the surprising effect on the queries with many loops (Figure 6(d)), where Basic-LRU-SP outperforms Karma when the entire data set fits in the aggregate cache. Karma pays for the use of DEMOTE, while Basic-LRU-SP enjoys "free" exclusive caching, along with the accurate priorities generated by Karma. The average I/O cost of Basic-LRU-SP is 92.5% lower than that of LRU. Karma's average I/O cost is 90.25% lower than that of LRU. When the aggregate cache is smaller than the data set, or when the pattern is not a pure loop (Figures 5, 6(a-c), 7, and 8), Karma makes optimal use of both caches and outperforms Basic-LRU-SP significantly.

How is Karma affected by the model parameters? We wanted to estimate Karma's sensitivity to varying access costs in different storage levels. Figure 9 shows how Karma behaves on query set 1 (the behavior is similar for all traces) when the disk access delay ranges from 10 to 100 times the delay of a READ from the second level cache (or a DEMOTE to that cache). When the delay for a disk access is larger, the "penalty" for DEMOTE is less significant compared to the decrease in the number of disk accesses. When DEMOTE is only ten times faster than a disk access, its added cost outweighs its benefits in very small caches.

How is Karma affected by the cache size at each level? Figure 10 compares the behavior of Karma to other policies on cache hierarchies with varying lower-level sizes. The details are numerous and so we present here only results for the best LRU-based policies, Double-2Q and Global-Multi-Q. Global-Multi-Q

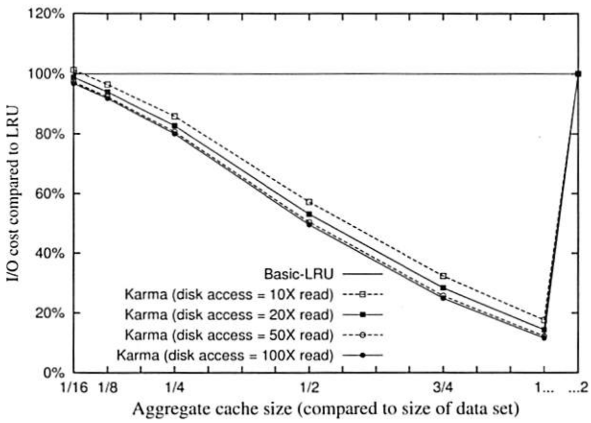


Figure 9: Karma's I/O cost compared to that of LRU for different disk access delays, for query set 1. When delays for a disk access are longer, there is more benefit in using DEMOTE, despite its additional cost.

performs futile DEMOTEs both when the cache sizes are small and when the lower cache is larger than the upper cache. In the first case, demoted blocks are discarded before being used, and in the second they are already present in the lower cache. As a result of data redundancy between caches, Double-2Q is highly sensitive to the portion of the aggregate cache that is in the upper level. Karma does not suffer from those problems and outperforms both policies (and those omitted from the graphs) in all cache settings.

We expect Karma to maintain its advantage when the difference in cache sizes increases. In the Basic and Double policies the benefit of the smaller cache will become negligible due to data redundancy, whereas in the Global policies the amount of futile Demote operations will increase. Karma, on the other hand, benefits from any additional cache space, in any level, and maintains exclusive caching by using only the necessary amount of DEMOTEs.

7 Related Work

We discussed several existing cache replacement policies in Section 5. Here we elaborate on additional aspects of related work.

Multilevel. Wong and Wilkes [49] introduced the DEMOTE operation to prevent inclusion between cache levels. They assume that network connections are much faster than disks. In such settings, performance gains are still possible even though a DEMOTE may incur a network cost. Cases where network bandwidth is the bottleneck instead of disk contention were addressed in a later study [51]. There, instead of evicted blocks being demoted from the first to the second level cache, they are reloaded (prefetched) into the second level cache from the disk. A complementary approach has the application attach a "write hint" [29] to each WRITE command, choosing one of a few reasons for performing the write. The storage cache uses these hints to "guess" which blocks were evicted from the cache and which are cached for further use. In X-Ray [45] the information on the content of the upper level cache is obtained using gray-box techniques, and derived from operations such as file-node and write log updates.

In ULC [24], the client cache is responsible for the content of all cache levels underneath. The level of cache in which blocks should be kept is decided by their expected locality of reference. A "level tag" is attached to each I/O operation, stating in which cache level the block should be stored. In *heterogeneous caching* [4], some degree of exclusivity can be achieved without any cooperation between the levels, when each cache level is managed by a different policy. A multilevel cache hierarchy is presented, where each cache is managed by an adaptive policy [20], ACME (Adaptive Caching using Multiple Experts): the "master" policy monitors the miss rate of a pool of standard replacement policies and uses machine learning algorithms to dynamically shift between those policies, in accordance with changing workloads.

Karma uses READ-SAVE to avoid unnecessary DEMOTEs. As with the average read access time in ULC [24], our storage model enables specific calculations of the DEMOTE operations that actually occurred, instead of estimating them in the cost of each READ, as in the original evaluation [49]. The use of READ-SAVE reflects a non-centralized operation, as opposed to the level tags. Our results in Section 6 (Figures 5, 7, and 8) show that exclusive caching is not enough to guarantee low I/O cost. Karma is able to make informed replacement decisions to achieve better results.

Detection-based caching. Detection based policies use history information for each block in the cache (and sometimes for blocks already evicted from the cache) to try to "guess" the access pattern of the application. This may help identify the best candidate block for eviction. DEAR [13], AFC [14], and UBM [27], which are designed for a file system cache, all collect data about the file the block belongs to or the application requesting it, and derive access patterns (sequential, looping, etc.).

In PCC [19], the I/O access patterns are correlated with the program counter of the call instruction that triggers the I/O requests, enabling differentiation between multiple patterns in the same file if they are invoked by different instructions. Each pattern is allocated a partition in the cache, whose size is adjusted dynamically until the characterization of patterns stabilizes.

MRC-MM [50] monitors accesses to virtual memory pages in order to track the page miss ratio curve (MRC) of applications. The MRC in each "epoch" is then used

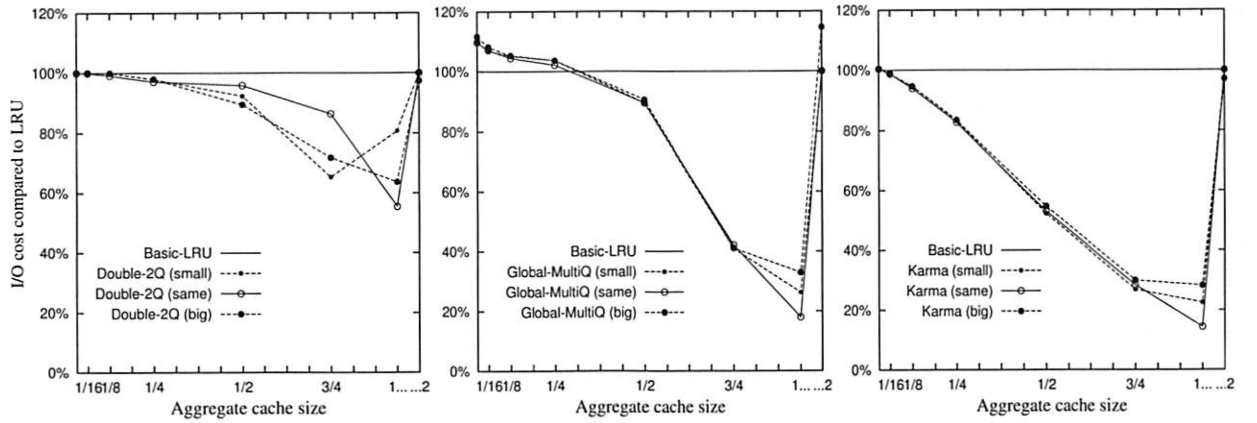


Figure 10: Weighted I/O cost on query set 1 (the behavior is similar for all traces) when cache levels are of different sizes. For lack of space we show results only for Karma and for the best LRU-based policies. *Same:* $|Cache_2| = |Cache_1|$. *Big:* $|Cache_2| = |Cache_1| \times 2$. *Small:* $|Cache_2| = |Cache_1|/2$. Double-2Q is highly sensitive to the difference in size and Global-MultiQ suffers from overhead of DEMOTE, while Karma outperforms all policies for all cache settings.

to calculate the marginal gain of all processes. Memory utilization is maximized by allocating larger memory portions to processes with higher marginal gain.

SARC [17], designed for a second level cache, detects sequential streams based on access to contiguous disk tracks. This information is used for sequential prefetching and cache management. The cache is dynamically partitioned between random and sequential blocks.

Since Karma is provided with the characterization of block ranges, it does not incur the additional overhead of gathering and processing access statistics. The access pattern and access frequency for each range are known and on-the-fly adjustment of partition sizes is avoided. Karma adjusts its partitions only in transitional phases, when the application changes its access pattern.

Informed caching. A different approach to determining access patterns is to rely on application hints that are passed to the cache management mechanism. This eliminates the need for detection, thus reducing the complexity of the policy. However, relying on hints admittedly limits applicability. Existing hint based policies require the applications to be explicitly altered to manage the caching of their own blocks. LRU-SP [10] and TIP2 [38] are two such policies.

In TIP2, applications disclose information about their future requests via an explicit access string submitted when opening a file. The cache management scheme balances caching and prefetching by computing the value of each block to its requesting application.

Unlike TIP2, Karma does not require an explicit access string, but a general characterization of the observed access patterns (i.e., looping, random or sequential). In this way, it is similar to LRU-SP. This makes Karma useful for applications such as databases, where accesses can be characterized in advance into patterns.

The ability of databases to disclose information about future accesses has made them ideal candidates for hint generation. Database query optimizers [44] choose the optimal execution path for a query. They aim to minimize the execution cost, which is a weighted measure of I/O (pages fetched) and CPU utilization (instructions executed). Once the optimal path is chosen, the pattern of access to the relevant blocks is implicitly determined. It can then be easily disclosed to the cache manager. A fundamental observation [47] was that in order for an operating system to provide buffer management for database systems, some means must be found to allow it to accept "advice" from an application program concerning the replacement strategy. The following studies base their replacement policy on this method.

A *hot set* is a set of pages over which there is a looping behavior [41]. Its size can be derived from the query plan generated by the optimizer. In the derived replacement policy [41], a separate LRU queue is maintained for each process, with a maximal size equal to its hot set size. DBMIN [15] enhances the hot set model in two ways. A hot set is defined for a file, not for an entire query. Each hot set is separately managed by a policy selected according to the intended use of the file.

By adding marginal gains to this model, MG-x-y [36] is able to compare how much each reference string will "benefit" from extra cache blocks. The marginal gain of random ranges is always positive, and so MG-x-y avoids allocating the entire cache to only a small number of such ranges by imposing a maximum allocation of y blocks to each of the random ranges.

Karma builds upon the above policies, making a fine-grained distinction between ranges. A file may contain several ranges accessed with different characteristics. As in DBMIN and MG-x-y, each range is managed with a policy suited for its access pattern. Like MG-x-y, Karma uses marginal gains for allocation decisions, but instead of limiting the space allocated to each range it brings every accessed block into $Cache_1$ to capture fine-grain locality. Most importantly, unlike the above policies, Karma maintains all its benefits over multiple cache levels.

A recent study [11] evaluates the benefit of *aggressive collaboration*, i.e., use of DEMOTE, hints, or level tags, over *hierarchy-aware* caching, which does not require modifications of current storage interfaces. Their results show that the combination of hints with global management yields only a slight advantage over the hierarchy aware policies. However, they experiment with very basic hints, combined with LRU or ARC management, while it is clear from our results that simply managing loops with MRU replacement is enough to achieve much better results. Since Karma distinguishes between access patterns and manages each partition with the policy best suited for it, its improvement is significant enough to justify the use of hints.

Storage system design. Modern storage systems are designed as standalone platforms, separated from their users and applications by strict protocols. This modularity allows for complex system layouts, combining hardware from a range of manufacturers. However, the separation between the storage and the application layers precludes interlayer information sharing that is crucial for cooperation between the systems components - cooperation we believe will lead to substantial performance gains.

Many recent studies attempt to bypass this inherent separation: the levels gather knowledge about each other by tracking the implicit information exposed by current protocols. For example, in several recent studies [32, 43] the operating system extracts information about the underlying disk queues and physical layout from disk access times. In the gray-box approach [5, 46], the storage system is aware of some operating system structures (such as inodes), and extracts information about the current state of the file system from operations performed on them. In C-Miner [30], no knowledge is assumed at the storage level. The storage system uses data mining techniques in order to identify correlations between blocks. The correlations are then used for improving prefetching and data layout decisions.

In contrast, other studies explore the possibility of modifying traditional storage design. Some offer specific protocol additions, such as the DEMOTE operation [49]. Others suggest a new level of abstraction, such as object storage [6]. Other work focused on introducing new storage management architectures aimed at optimizing database performance [21, 42].

Karma requires some modification to existing storage interfaces, which is not as substantial as that described above [6, 21, 42]. This modification will enable the storage system to exploit the information available in other levels, without paying the overhead of extracting and deriving this information. We believe that the additional complexity is worthwhile, given the large performance benefits presented in Section 6.

Policy	1st level	2nd level	Extra Information	Exclusive
LRU	✓	✓	X	X
DEAR,UBM AFC,PCC	✓	X	file,application PC	X
LRU-SP	✓	X	pattern,priority	X
TIP2	✓	X	explicit trace	X
HotSet,DBMIN MG-x-y	✓	X	query plan	X
X-Ray Write hints	X	✓	indirect hints	✓
2Q,MultiQ ARC,CAR,SARC	X	✓	X	X
Demote Global-L2	X	✓	X	✓
ULC	✓	✓	client instructions	✓
ACME	✓	✓	(machine learning)	✓
Karma	✓	✓	ranges	✓

Table 2: Summary of cache management policies, the extra information they require, and whether or not they are suitable for use in first and second level caches.

Table 2 summarizes the policies discussed in this paper. The policies are compared according to their ability to perform well in more than one cache level, to achieve exclusiveness in a multilevel cache system, and to use application hints.

8 Conclusions and Future Work

We defined a model for multilevel caching and defined the weighted I/O cost of the system for this model as the sum of costs of all operations performed on a trace. We propose a policy which solves the three problems that can occur in a multilevel cache: blurring of locality of reference in lower level caches, data redundancy, and lack of informed caching at the lower cache levels. None of the existing policies address all these problems.

Our proposed policy, Karma, approximates the behavior of the optimal offline algorithm, MIN. Like MIN, it aims to optimize the cache content by relying on knowledge about the future, instead of on information gathered in the past. Karma uses application hints to partition the cache and to manage each range of blocks with the policy best suited for its access pattern. It saves in the cache the blocks with the highest marginal gain and achieves exclusive caching by partitioning the cache using DEMOTE and READ-SAVE. Karma improves the weighted I/O cost of the system significantly. For example, on a permutation of TPC-H queries, Karma improves over pure LRU by an average of 85%. It adds an average

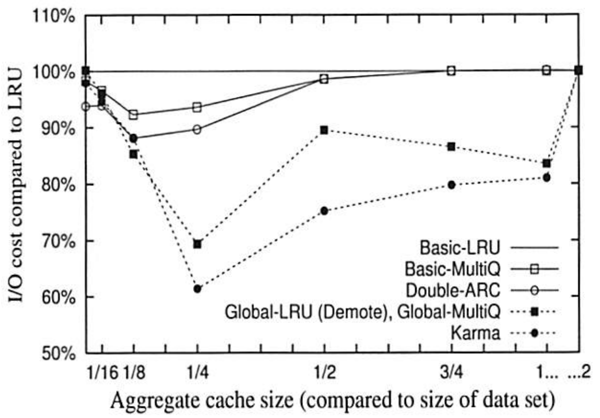


Figure 11: Preliminary results for an OLTP workload. Using TPCC-UVa [31], an open-source implementation of the TPC-C [3] benchmark, we created a trace of over 10,000,000 I/Os accessing 130,000 distinct blocks. The hints for Karma were generated using the "explain" output for each transaction's queries, and the frequency of each transaction. We present only the best LRU based policies. Karma's improvement over LRU is greater than the improvement of all of these policies, in all cache sizes but one, where it equals the improvement of Global-MultiQ.

of 50% to the improvement of Demote over LRU and an average of 25% to that of LRU-SP.

When more features are added to Karma, we believe it will be able to achieve such improvement on workloads that are essentially different from decision support. We intend to add calculations of marginal gain for random ranges which are not necessarily of uniform distribution. Karma will also handle ranges which are accessed concurrently with more than one access pattern. Figure 11 shows our initial experiments with an OLTP workload, demonstrating that even without such additions, Karma outperforms existing LRU-based algorithms on such a trace by as much as 38%.

The framework provided by Karma can be extended to deal with further additions to our storage model. Such additions may include running multiple concurrent queries on the same host, multiple caches in the first level, or prefetching. DEMOTE and READ-SAVE can still be used to achieve exclusiveness, and the marginal gains will have to be normalized according to the new parameters. Karma relies on general hints and does not require the application to submit explicit access strings or priorities. Thus, we expect its advantages will be applicable in the future not only to databases but to a wider range of applications.

Acknowledgments

We thank our shepherd Scott Brandt and the anonymous reviewers for their helpful comments. We also thank Kai Li and Avi Yadgar for fruitful discussions.

References

[1] Working draft SCSI block commands - 2 (SBC-2), 2004.

[2] TPC benchmark H standard specification, Revision 2.1.0.

[3] TPC benchmark C standard specification, Revision 5.6.

[4] Ismail Ari. *Design and Management of Globally Distributed Network Caches.* PhD thesis, University of California Santa Cruz, 2004.

[5] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Symposium on Operating Systems Principles (SOSP)*, 2001.

[6] Alain Azagury, Vladimir Dreizin, Michael Factor, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, Julian Satran, Ami Tavory, and Lena Yerushalmi. Towards an object store. In *NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2003.

[7] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *USENIX Conference on File and Storage Technologies (FAST)*, 2004.

[8] Tsipora Barzilai and Gala Golan. Accessing application identification information in the storage tier. Disclosure IL8-2002-0055, IBM Haifa Labs, 2002.

[9] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[10] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[11] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *SIGMETRICS*, 2005.

[12] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based placement for storage caches. In *USENIX Annual Technical Conference*, 2003.

[13] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. An implementation study of a detection-based adaptive block replacement scheme. In *USENIX Annual Technical Conference*, 1999.

[14] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *SIGMETRICS*, 2000.

[15] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *International Conference on Very Large Data Bases (VLDB)*, 1985.

[16] Gregory R. Ganger, Daniel Ellard, and Margo I. Seltzer. File classification in self-∗ storage systems. In *International Conference on Autonomic Computing (ICAC)*, 2004.

[17] Binny S. Gill and Dharmendra S. Modha. SARC: Sequential prefetching in adaptive replacement cache. In *USENIX Annual Technical Conference*, 2005.

[18] Binny S. Gill and Dharmendra S. Modha. WOW: Wise ordering for writes-combining spatial and temporal locality in non-volatile caches. In *USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[19] Chris Gniady, Ali R. Butt, and Y. Charlie Hu. Program counter based pattern classification in buffer caching. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[20] Robert B. Gramacy, Manfred K. Warmuth, Scott A. Brandt, and Ismail Ari. Adaptive caching by refetching. In *Neural Information Processing Systems (NIPS)*, 2002.

[21] Stavros Harizopoulos and Anastassia Ailamaki. A case for staged database systems. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.

[22] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[23] Song Jiang and Xiaodong Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, 2002.

[24] Song Jiang and Xiaodong Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[25] Theodore Johnson and Dennis Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *International Conference on Very Large Data Bases (VLDB)*, 1994.

[26] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file system (ELFS): an object-oriented approach to high performance file I/O. In *OOPSLA*, 1994.

[27] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[28] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *SIGMETRICS*, 1999.

[29] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[30] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-miner: Mining block correlations in storage systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2004.

[31] Diego R. Llanos and Belén Palop. An open-source TPC-C implementation for parallel and distributed systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[32] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. In *USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[33] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[34] Bruce Momjian. *PostgreSQL: Introduction and Concepts.* Addison-Wesley, 2000.

[35] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. In *USENIX Winter Conference*, 1992.

[36] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible buffer allocation based on marginal gains. In *ACM SIGMOD International Conference on Management of Data*, 1991.

[37] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *ACM SIGMOD International Conference on Management of Data*, 1993.

[38] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 224–244. IEEE Computer Society Press and Wiley, NY, 2001.

[39] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS*, 1990.

[40] Giovanni Maria Sacco and Mario Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *International Conference on Very Large Data Bases (VLDB)*, 1982.

[41] Giovanni Maria Sacco and Mario Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems (TODS)*, 11(4), 1986.

[42] Jiri. Schindler, Anastassia Ailamaki, and Gregory R. Ganger. Lachesis: Robust database storage management based on device-specific performance characteristics. In *International Conference on Very Large Data Bases (VLDB)*, 2003.

[43] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[44] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *ACM SIGMOD International Conference on Management of Data*, 1979.

[45] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[46] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[47] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[48] Masamichi Takagi and Kei Hiraki. Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches. In *International Conference on Supercomputing (ICS)*, 2004.

[49] Theodore M. Wong and John Wilkes. My cache or yours? Making storage more exclusive. In *USENIX Annual Technical Conference*, 2002.

[50] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[51] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, 2004.

AMP: Adaptive Multi-stream Prefetching in a Shared Cache

Binny S. Gill and Luis Angel D. Bathen
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120
Emails: binnyg@us.ibm.com, lbathen@uci.edu

Abstract— Prefetching is a widely used technique in modern data storage systems. We study the most widely used class of prefetching algorithms known as *sequential prefetching*. There are two problems that plague the state-of-the-art sequential prefetching algorithms: (i) *cache pollution*, which occurs when prefetched data replaces more useful prefetched or demand-paged data, and (ii) *prefetch wastage*, which happens when prefetched data is evicted from the cache before it can be used.

A sequential prefetching algorithm can have a fixed or adaptive degree of prefetch and can be either synchronous (when it can prefetch only on a miss), or asynchronous (when it can also prefetch on a hit). To capture these distinctions we define four classes of prefetching algorithms: Fixed Synchronous (FS), Fixed Asynchronous (FA), Adaptive Synchronous (AS), and Adaptive Asynchronous (AA). We find that the relatively unexplored class of AA algorithms is in fact the most promising for sequential prefetching. We provide a first formal analysis of the criteria necessary for optimal throughput when using an AA algorithm in a cache shared by multiple steady sequential streams. We then provide a simple implementation called AMP, which adapts accordingly leading to near optimal performance for any kind of sequential workload and cache size.

Our experimental set-up consisted of an IBM xSeries 345 dual processor server running Linux using five SCSI disks. We observe that AMP convincingly outperforms all the contending members of the FA, FS, and AS classes for any number of streams, and over all cache sizes. As anecdotal evidence, in an experiment with 100 concurrent sequential streams and varying cache sizes, AMP beats the FA, FS, and AS algorithms by 29-172%, 12-24%, and 21-210% respectively while outperforming OBL by a factor of 8. Even for complex workloads like SPC1-Read, AMP is consistently the best performing algorithm. For the SPC2 Video-on-Demand workload, AMP can sustain at least 25% more streams than the next best algorithm. Finally, for a workload consisting of short sequences, where optimality is more elusive, AMP is able to outperform all the other contenders in overall performance.

I. Introduction

Over the last several decades, we have witnessed remarkable improvements in the information processing capabilities of computing systems. A large number of data storage technologies have also been developed with diverse speeds, capacities, reliability and affordability characteristics. We often find that cost considerations force us to design systems with a data storage component which runs significantly slower than the processing unit. To bridge this gap between the data supplier and the data consumer, faster data caches are placed between the two. Since caches are expensive, they can typically keep only a subset of the entire data-set. Consequently, it is extremely important to manage the cache wisely in order to maximize its performance. The cornerstone of read cache management is to keep recently requested data in the cache in the hope that such data will be requested again in the near future. Data is placed in the cache only when requested by the consumer (*demand-paging*). Another, and rather competing method, is to fetch into the cache data that is predicted to be requested in the near future (*prefetching*).

A. Where is Prefetching Applied

The technique of prefetching dates as far back as the mid-sixties when multiple words were prefetched in processors in the form of a cache line. It was soon realized that increasing the size of the cache line can decrease performance due to false sharing. So, numerous hardware-initiated prefetching techniques were introduced in both uniprocessor and multiprocessor architectures [1], [2], [3]. Subsequently, software-initiated methods for prefetching were introduced where applications disclosed access patterns to the hardware or controlled prefetching directly [4], [5]. For other applications, compiler techniques were used to predict access patterns and insert fetch requests in the compiled executables [6], [7]. Compiler-assisted prefetching was also extended for pointer-based accesses [8], [9], [10].

Today prefetching is ubiquitously applied in web servers and clients [11], databases [12], file servers [13], [14], on-disk caches [15], and multimedia servers [16].

B. When is Prefetching Useful

The goal of prefetching is to make data available in the cache before the data consumer places its request, thereby masking the latency of the slower data source below the cache. However, prefetching is not without cost. It requires (i) cache space to keep the prefetched data; (ii) network bandwidth to transfer the data to the cache; (iii) data source bandwidth to read the data; and (iv) processing power to carry out the prefetch. If the prefetched data is not subsequently used by the data consumer, the extra cost of prefetching normally reduces performance. Only in over-provisioned systems, can prefetching with low predictive accuracy improve performance. However, the data cache is obviously under-provisioned as it can keep only a subset of the data-set. The prefetched data typically shares the cache space with demand-paged data. Therefore, the utility of the prefetched data should not be lower than the utility of the demand-paged data it replaces. To maximize the performance, the marginal utility of both kinds of data should be equalized [17]. Since the utility of prefetched data that is not subsequently used is zero, it is extremely important to prefetch judiciously, keeping the number of wasted prefetches to a minimum. Furthermore, any prefetching algorithm needs to be able to predict accesses sufficiently in advance to allow for the time it takes to prefetch the data. As a rule of thumb, prefetching is useful when the long-term prediction accuracy of access patterns is high.

C. What to Prefetch

The most common prefetching approach is to perform sequential readahead. The simplest form is One Block Lookahead (OBL), where we prefetch one block beyond the requested block [18]. OBL can be of three types: (i) *always prefetch* – prefetch the next block on each reference, (ii) *prefetch on miss* – prefetch the next block only on a miss, (iii) *tagged prefetch* – prefetch the next block only if the referenced block is accessed for the first time. *P-Block Lookahead* extends the idea of OBL by prefetching P blocks instead of one, where P is also referred to as the *degree of prefetch*. Dahlgren [19] proposed a version of the P-Block Lookahead algorithm which dynamically adapts the degree of prefetch for the workload. Tcheun [20] suggested a per stream scheme which selects the appropriate degree of prefetch on each miss based on a prefetch degree selector (PDS) table. For the case where cache is abundant, *Infinite-Block Lookahead* has also been studied [21].

Stride-based prefetching has also been studied mainly for processor caches where strides are detected based on information provided by the application [22], a lookahead into the instruction stream [23], or a reference prediction table indexed by the program counter [24]. Dahlgren [25] found that sequential prefetching is a better choice because most strides lie within the block size and it can also exploit locality.

History-based prefetching has been proposed in various forms. Grimsrud [26] uses a history-based table to predict the next pages to prefetch. Prefetching using Markov predictors has been studied in [27], wherein multiple memory predictions are prefetched at the same time. Data compression techniques have also been applied to predict future access patterns [12]. Vitter [28] provided an optimal (in terms of the miss ratio) prefetching technique based on the Lempel-Ziv algorithm. Lei [29] suggested a file prefetching technique based on historical access correlations maintained in the form of access trees.

The fact is, most commercial data storage systems use very simple prefetching schemes like sequential prefetching. This is because only sequential prefetching can achieve a high long-term predictive accuracy in data servers. Strides that cross page or track boundaries are uncommon in workloads and therefore not worth implementing. History-based prefetching suffers from low predictive accuracy and the associated cost of the extra reads on an already bottlenecked I/O system. The data storage system cannot use most hardware-initiated or software-initiated prefetching techniques as the applications typically run on external hardware. Further, offline algorithms [30], [31], [32], [33] are not applicable as they require knowledge of future data accesses.

D. The Problem of Cache Pollution

In the context of prefetching, *cache pollution* is said to occur when prefetched data replaces more useful data (demand-paged or prefetched) from the cache. There have been attempts to reduce cache pollution by restricting the amount of cache the prefetched data can occupy [34], or via software hints [35]. The SARC algorithm [17] provides an adaptive and autonomous solution to limit this problem by allocating cache space so as to equalize the marginal utility of the demand paged and prefetched data. However, we are not aware of any prior online solution for minimizing cache pollution that occurs when new prefetched data replaces more useful prefetched data from the cache.

E. The Problem of Wasted Prefetches

In data storage systems, the disks are typically the bottleneck. If pages are prefetched speculatively and are not subsequently used, then not only does this cause cache pollution and increase in the backend bandwidth usage, but more importantly, it causes additional I/O load on the disks. This additional load can lead to degradation in performance, defeating the purpose of prefetching. This is the reason why most history-based prefetching schemes which do not have high prediction accuracy are not used in commercial systems.

F. Our Contributions

A prefetching algorithm can have a fixed or adaptive degree of prefetch and can be either asynchronous (when it can prefetch on a hit) or synchronous (when it can prefetch only on a miss). This naturally leads to four classes which we call Fixed Synchronous (FS), Fixed Asynchronous (FA), Adaptive Synchronous (AS), and Adaptive Asynchronous (AA).

Although sequential and non-sequential data typically occupy the same cache, it is worthwhile to examine the prefetched data alone as most known prefetching algorithms suffer from cache pollution and prefetch wastage, and thus, can be improved.

We examine the case where an LRU (Least Recently Used) cache houses prefetched data for multiple concurrent sequential streams. We provide a theoretical analysis and prove the sufficient conditions for optimal online cache management for steady-state sequential streams. We also provide a simple implementation called AMP, the first member of the AA class which optimally adapts both the degree of prefetch and the timing thereof according to the workload and cache size constraints.

With a theoretically optimal design, AMP minimizes prefetch wastage and cache pollution within the prefetched data while maximizing the aggregate throughput achieved by the sequential streams. To demonstrate the effectiveness of AMP, we compare it with 9 other prefetching algorithms including the best representatives from the FA, FS, and AS classes, over a wide range of cache sizes, request rates, request sizes, number of concurrent streams, and workloads.

We observe that AMP convincingly outperforms all the FA, FS, and AS algorithms for any number of streams, and over all cache sizes. In an experiment with a 100 concurrent sequential streams and varying cache sizes, AMP beats the FA, FS, and AS algorithms by 29-172%, 12-24%, and 21-210% respectively while outperforming no prefetching and OBL by a factor of 8. AMP is consistently the best performing algorithm in both the small cache and large cache scenarios, even for complex workloads like SPC1-Read. For SPC2 Video-on-Demand workload, AMP can support at least 25% more streams than the next best algorithm. For streams with short sequences, as well, for which optimality is more elusive, AMP surpasses all the other contenders in its overall performance.

G. Outline of the Paper

In Section II, we suggest a useful classification of sequential prefetching algorithms and examine each in detail. In Section III, we provide a formal analysis and proof for the conditions necessary for optimal sequential prefetching. We also provide an implementation called AMP. In Section IV, we describe the workloads used in this paper. In Section V, we delineate the experimental setup used for our experiments. In Section VI, we present the experimental results and conclude with our findings in Section VII.

II. Sequential Prefetching

A. Rules of engagement

The cost of caching an entity is equal to the size of the entity multiplied by the amount of time for which it is present in the cache. The benefit of caching an entity, on the other hand, is the number of hits it produces. We define two self-evident rules that any prefetching algorithm should follow:

- *Avoid Wastage Rule*: Do not prefetch any page that will be evicted before it is requested by the workload.
- *Avoid Cache Pollution Rule*: Do not prefetch any page that will evict other prefetched pages without providing any net gain in performance.

B. Synchronous Vs. Asynchronous Prefetching

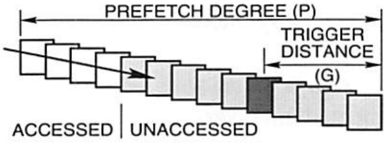


Fig. 1. Asynchronous Prefetching

There are two kinds of prefetch requests: (i) *synchronous* prefetch, and (ii) *asynchronous* prefetch. A *synchronous* prefetch is when on a *miss* on page x, we prefetch p extra pages beyond page x. It merely extends the extent of the client's read request to include more pages. On the other hand, an *asynchronous* prefetch is when on a cache *hit* on a page x, we create a new read request to prefetch p pages beyond those already in the cache. In each set of p prefetched pages, a *trigger* page is identified at a *trigger distance* of g from the end of the prefetched set of pages (Figure 1). When $g = 0$, the trigger is set on the last page of the prefetched set. When a *trigger* page is hit, an asynchronous prefetch is requested for the next set of p sequential pages. Unlike synchronous prefetching, asynchronous prefetching enables us to always stay ahead of sequential read requests and for suitable values of p and g, never incur a read miss after the initial miss for a sequential stream [17]. Asynchronous prefetching is always used in conjunction with some form of synchronous prefetching to prefetch the initial set of pages.

Notice that asynchronous prefetching creates new read requests on its own and, therefore, in cases where prefetches are wasted, asynchronous prefetching will have more disk seeks on the backend for the same workload than synchronous prefetching. However, for larger prefetches on data striped across disks, even synchronous prefetches will result in new read requests.

As a guideline, asynchronous prefetching should be avoided in cases where prefetch wastage is high.

C. A Classification of Prefetching Algorithms

Sequential prefetching is the most promising and widely deployed prefetching technique for data servers. It has a high predictive accuracy and is extremely simple to implement. Simple methods are used to isolate the sequential components of workloads [17], upon which prefetching is applied. We can classify the known sequential prefetching techniques as follows:

- *Fixed Synchronous (FS) prefetching*: The simplest form of sequential prefetching is where we prefetch the next page on a miss (OBL [18]). Another variant is where a fixed number of pages (p) are prefetched on every miss.
- *Adaptive Synchronous (AS) prefetching*: This is a popular form of sequential prefetching where the number of pages prefetched on every miss (p) is gradually increased as the length of the sequence referenced becomes longer. The degree of prefetch, p starts with 1 and is either linearly incremented on every miss [20] *(Linear-AS)* or exponentially incremented *(Exp-AS)*. Usually, there is a predefined upper limit for incrementing p. Although *Exp-AS* adapts faster than the *Linear-AS* method, it is prone to more wastage in workloads with many short sequences or when cache space is limited.
- *Fixed Asynchronous (FA) prefetching*: In this class, a hit on a trigger page causes a prefetch. Tagged prefetching [18], a variant of OBL, is the earliest example of asynchronous prefetching where the degree of prefetch was 1 and the trigger distance was 0. Subsequently, the idea has been extended to any pair of fixed values of p and g [17]. Unlike synchronous prefetching methods, this class of algorithms can achieve zero misses for a workload when the chosen values of p and g are adequate. However, since the algorithm is hand-tuned and not adaptive, it does not work well for all workloads.
- *Adaptive Asynchronous (AA) prefetching*: To the best of our knowledge, there is no published work that dynamically adapts both the degree of prefetch (p) and the trigger distance (g). This is the most promising class of prefetching algorithms. The only algorithm that comes close is the one proposed by Dalhgren [19] where the degree of prefetch is the same for all streams and every page is a trigger page. It is not truly applicable in the context of data servers because it is wasteful. As each page is a trigger page it inefficiently prefetches one page at a time for sequential streams. It also *requires* some amount of prefetch wastage to adapt p which is blindly applied for all sequential streams.

D. Interaction between demand-paged and prefetched data

Since demand-paged data, prefetched data, and sometimes modified data, share the same cache in most data server architectures, we normally would require a way to divide the cache between the various types, and manage each portion optimally, so as to maximize the overall performance of the system. While a large number of demand-paging cache replacement algorithms have been devised (for example, LRU, CLOCK, FBR, 2Q, LRFU, LIRS, MQ, and ARC), surprisingly, and to the best of our knowledge, there has been no research towards an online optimal cache replacement policy for prefetched data.

In this paper, we provide this missing link and present a provably optimal algorithm for multiple sequential streams sharing a cache and a very simple practical implementation thereof. We believe that much of the work on understanding the interactions between various types of cached data ([4], [17], [30], [33]) will benefit from and incorporate our algorithm and analysis.

III. AMP

A. Replacement Policy for Prefetched data

The most widely used data structure for cache replacement policy is LRU, mainly because of its simplicity. This policy leverages temporal locality in the workload to improve cache hit ratios. Even within sequentially prefetched data, it is possible to have non-sequential accesses that exhibit temporal locality. This has encouraged most commercial systems to use the LRU data structure and replacement policy even for prefetched data rather than simply evicting prefetched data immediately after use. In this paper, we improve this LRU policy by making it aware of the difference between prefetched and demand-paged data. A prefetched page is moved to the most recently used (MRU) position only on repeated access and not on the first access.

B. Theoretical Analysis: Optimality Criteria

In this section we theoretically analyze the case when an LRU cache is shared by prefetched data for multiple steady sequential streams. We assume an AA algorithm operating with a cache of size C. L is defined as the average life of a page in the cache. Each stream has a maximum request rate, r, which is achieved when all read requests are hits in the cache. The aggregate throughput, B, is the sum of individual stream throughputs ($B = \sum_i b_i$).

Observation III.1 *$p/t(p)$ is a monotonically non-decreasing function, where $t(p)$ is the average time to prefetch p pages.*

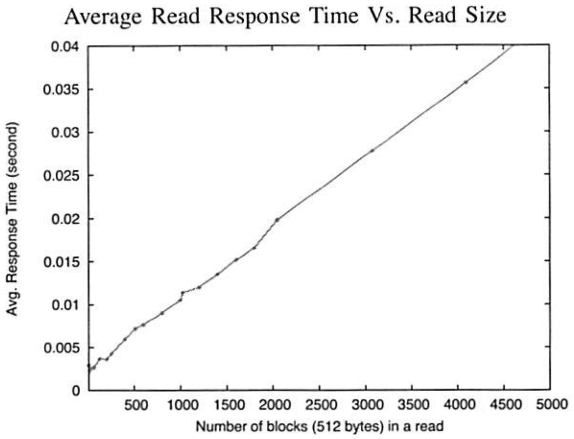


Fig. 2. Starting at 3 ms, the average response time grows linearly with the read size. We observed similar behavior for RAID-5 implying that the optimality criteria in this paper apply to RAID as well.

Proof: From Figure 2 we observe that $t(p)$ is of the form $kp + c$. Hence, $p/t(p) = \frac{p}{kp+c}$, and its slope

$$\frac{db}{dp} = \frac{c}{(kp+c)^2}$$

is positive since c is positive. ∎

Definition III.1 *A stream is said to be* satisfied *at* (p, g), *if it experiences no misses for the given* p *and* g.

Lemma III.1 *A stream is* satisfied at (p, g) *iff* $t(p) \leq (g + 1)/r$.

Proof: By definition, if a stream is satisfied at (p, g) then it experiences no misses in the steady state. That implies that the prefetch issued at the trigger distance g completes before the $g+1$ pages are read by the client. Therefore, $t(p) \leq (g + 1)/r$, where r is the request rate of the stream. The reverse is also true. If the time it takes to prefetch p pages is not more than the time it takes the client to read the g pages, then there cannot be any misses in the steady state implying that the stream is satisfied. ∎

Observation III.2 *The throughput of a satisfied stream is equal to* r, *its request rate.*

Proof: By definition, a satisfied stream experiences no misses in the steady state. No misses implies no stall time and the stream proceeds at its desired request rate of r. ∎

Lemma III.2 *Cache Pollution occurs if* $(g + 1) > \lceil r \cdot t(p) \rceil$.

Proof: If $g + 1 > \lceil r \cdot t(p) \rceil$ then $g \geq r \cdot t(p)$ or $t(p) \leq g/r < (g + 1)/r$. By Lemma III.1, the stream is satisfied at the chosen (p, g) but is also satisfied for $g - 1$ at $(p, g - 1)$. By Observation III.2, the throughput with $g - 1$ will remain the same as the stream will remain satisfied. However, the cost of the case where a lower g is used is smaller as the average number of pages that have to be kept in the cache is smaller. Hence, cache space is being wasted without any gain in throughput. ∎

Lemma III.3 *If there is no cache pollution, wastage occurs iff* $p/r > L$.

Proof: If there is no cache pollution, $(g + 1) \leq \lceil r \cdot t(p) \rceil$ (Lemma III.2). By their definitions, p pages are requested when $g + 1$ pages from the previous prefetch are still unaccessed. The number of these pages that are consumed in the time it takes to prefetch is $r \cdot t(p)$, which is roughly all of the unaccessed pages. Hence, as soon as the next p pages are prefetched, they begin to be consumed at the request rate r. Therefore, the time it takes for the p pages to be consumed after prefetch is p/r. Now, if the average life (L) of pages in the cache is less than p/r, then some of the prefetched pages will be evicted before they are requested. Conversely, if L is greater than p/r then all the p pages will be requested before they reach the LRU end of the list and face eviction. ∎

Lemma III.4 *If there is no cache pollution, throughput of a stream* $(b) = min(r, \frac{p}{t(p)}, \frac{r \cdot L}{t(p)})$.

Proof: The throughput of a stream cannot exceed its request rate (r). Further, since we use a single outstanding prefetch for a stream at any time, the throughput cannot exceed the amount prefetched (p) divided by the time it takes to prefetch that amount $t(p)$. In the case where $p > r \cdot L$, wastage occurs (Lemma III.3) and only $r \cdot L$ pages out of p will be accessed before being evicted. In this case, the throughput is limited by $r \cdot L/t(p)$. ∎

Lemma III.5 *If there is no wastage* $B \cdot L = C$

Proof: The life of the cache (L) is equal to the time it takes to insert C new pages in the top end of the cache. If there is no wastage, the rate of insertion in the cache is equal to the aggregate read throughput of all the streams (B). Therefore, $C/B = L$. ∎

Lemma III.6 *For a fixed choice of* $p_1, p_2, ..., p_n$, *and cache of size* C, *the aggregate throughput* (B) *is unique when there is no wastage.*

Proof: Suppose, the aggregate throughput(B) was not unique. Without loss of generality, we would have

$B' > B$, such that

$$B' = \sum_{i=1..n} min(r_i, p_i/t(p_i), r_i \cdot L'/t(p_i))$$

$$B = \sum_{i=1..n} min(r_i, p_i/t(p_i), r_i \cdot L/t(p_i))$$

Since there is no wastage, we have $p \leq r \cdot L$ for all streams. So, the only different term in the min expression is not significant. Thus, $B' = B$, which is contrary to our assumption. ■

Theorem III.1 *The aggregate throughput of n streams sharing a cache of size C with average cache life L, is maximized if* $\forall i, p_i = \lfloor r_i \cdot L \rfloor$.

Proof: Given n streams with request rates of $r_1, r_2, ...r_n$ and a cache of size C, let the throughput obtained by the choice: $\forall i, p_i^T = \lfloor r_i \cdot L \rfloor$ be B_T. The theorem claims that B_T is the maximum aggregate bandwidth obtainable (B_{max}) through any choice of p_i^T.

We will prove by contradiction. Let us assume that $B_T < B_{max}$. Therefore, there exists some choice of $p_1, p_2, ..., p_n$ such that the aggregate bandwidth of all streams is B_{max}.

Since wastage and cache pollution can never increase aggregate throughput, we assume, without loss of generality, that B_{max} is free from these inefficiencies.

If the choice of p_i is the same as that specified by this theorem, then by Lemma III.6, $B_T = B_{max}$, which is contrary to our assumption.

$$\therefore \exists i : p_i \neq \lfloor r_i \cdot L \rfloor \quad (1)$$

By Lemma III.3, it must be the case for B_{max} that

$$\forall i, p_i \leq \lfloor r_i \cdot L \rfloor \quad (2)$$

If follows from (1) and (2), without loss of generality, that $p_1 < \lfloor r_1 \cdot L \rfloor$.

Let us define a new set: $p_1^1, p_2^1, ..., p_n^1$, where $p_1^1 = \lfloor r_i \cdot L^1 \rfloor$, and $\forall i \neq 1, p_i^1 = p_i$. L^1 and B^1 are the new cache life and aggregate throughput values.

Since $B^1 \leq B_{max}$ (by defn. of B_{max}), $L^1 \geq L$ (Lemma III.5). By Lemma III.4, $\forall i \neq 1, b_i^1 \geq b_i$ as $p_i^1 = p_i$ and $L^1 \geq L$. By Observation III.1 and Lemma III.4, $b_1^1 \geq b_1$ as $p_1^1 > p_i$.

$$\therefore B^1 = \sum b_i^1 \geq \sum b_i = B_{max}$$

Since $B^1 \leq B_{max}$, it follows that $B^1 = B_{max}$.

By repeating the above procedure for every stream with $p_i < \lfloor r_i \cdot L \rfloor$, we will arrive at a set $p_1^n, p_2^n, ..., p_n^n$, where $\forall i, p_i^n = \lfloor r_i \cdot L^n \rfloor$ and $B^n = B_{max}$.

Since, the choice of p for each stream will then be the same for B^n and B_T, $B^n = B_T$ by Lemma III.6.

$$\therefore B^n = B_{max} = B_T$$

which contradicts our assumption that $B_T < B_{max}$. ■

C. AMP *Algorithm*

The AMP algorithm, which adapts to achieve the optimality criteria set in the previous section, is outlined in Figures 3 and 4. We now draw attention to the important portions of the algorithm and the logic behind the choices we have made.

We make a conscious effort to avoid a separate data structure to track the adapted values of p and g for each detected sequential stream. We store the value of p and g in the page data structure. This removes any restriction on the number of streams that can be tracked.

Lines 20-23 implement the synchronous prefetching component of the algorithm. The number of pages to be prefetched on a read miss is not fixed (as in FS algorithms) but is the adapted value of p stored in the metadata of the previous page.

Whenever the current p is greater than the Asynchronous Prefetch Threshold (APT), asynchronous prefetching is activated. APT is set to an empirically reasonable value of 4. A page at a distance of $APT/2$ from the last page prefetched is chosen as the prefetch trigger page and the tag is set (Lines 41, 49). When there is a hit on a tag page, the tag is reset and an asynchronous prefetch is initiated for p pages as specified in the last page of the current set (Lines 27-30).

Adapting the degree of prefetch (p): As per Theorem III.1, we desire to operate at a point where $p = r \cdot L$. If p is more than this optimal value, the last page in a prefetched set will reach the LRU end unaccessed. We give such a page another chance by moving it to the MRU position and setting the *old* flag (Line 53). Whenever an unaccessed page is moved to the MRU position in this way, it is an indication that the current value of p is too high, and therefore, we reduce the value of p (Line 56). In Lines 31-35, p is incremented by the $readsize$ (the size of read request is pages) whenever there is a hit on the $last$ page of a read set (pages read in the same I/O) which is also not marked *old*.

Adapting the trigger distance (g): The main idea is to increment g if on the completion of a prefetch we find that a read is already waiting for the first page within the prefetch set (readWaiting()). If g was larger, the prefetch would have been issued earlier, reducing or eliminating the stall time for the read. Thus, we increment g in Line 47. However, we also need to decrement g when p itself is being decremented (Line 57). This keeps $g = r \cdot t(p)$ as per Lemmas III.1, III.2.

Whenever we adapt the value of p and g we store the updated values in the last page that has been read into the cache for the sequence. This is located by the lastInSequence(x) method in Lines 11-19. The method simply returns the last page of the set that x belongs to, or if the next set of pages has also been prefetched,

```
DATA STRUCTURE AND FUNCTIONS:

struct page {
    off_t $addr$;         // addr of the page
    off_t $Laddr$;        // addr of the last page in read set
    bool $accessed$;      // whether page has been accessed
    bool $tag$;           // if page hit will initiate prefetch
    bool $old$;           // if page was given another chance
    short int $p$;        // prefetch degree
    short int $g$;        // trigger distance
}

lookup(off_t $addr$)
1:  if (page $addr$ is present in cache)
2:      return page
3:  endif
4:  return null

#define prev(x) lookup($x{\rightarrow}addr - 1$)
#define last(x) lookup($x{\rightarrow}Laddr$)
#define isLast(x) ($x{\rightarrow}addr == x{\rightarrow}Laddr$)

createPages(page[] s, page ** $last$, page ** $prev$)
5:  *$prev$ = prev(first page in $s$)
6:  *$last$ = the last page in $s$
7:  foreach $x$ in $s$; do
8:      create page $x$ in cache
9:      $x{\rightarrow}Laddr = (*last){\rightarrow}addr$
10: done

lastInSequence(page * $x$)
11: $l_1$ = $last(x)$
12: if (!$l_1$)
13:     return null
14: endif
15: if (!lookup($x{\rightarrow}Laddr + 1$))
16:     return $l_1$
17: elsif ($l_2$ = lookup($x{\rightarrow}Laddr + l_1{\rightarrow}p$))
18:     return $l_2$
19: endif
```

Fig. 3. AMP: Data structures and support functions

it returns the last page of the next set. The logic is to keep the adapted values of p and g in the page that is most recently added to the cache and is thus most likely to stay in the cache for the longest time. In the unlikely case where the adapted values of p and g are forgotten because of page evictions, we set p to the current prefetch size, and set g to half of p.

Since AMP adapts to discover the optimal values of p and g, it incurs a minor cost of adaptation which quickly becomes negligible and allows it achieve near optimal performance.

In the eviction part of the algorithm (Lines 50-59), pages that are *old* or *accessed* are evicted from the LRU end. Finally, we always make sure that $p \geq g+1$ (Lines 44, 57) and $p \leq p\_threshold$ (a reasonable 256 pages for all algorithms).

```
ON HIT OR MISS:

A read request of $readsize$ pages is processed one page
at a time as follows:

On read miss on page $x$
20: read page $x$ along with
21:     (a) remaining pages in read request
22:     (b) if ($prev(x)$)
23:             $prev(x){\rightarrow}p$ pages beyond the read request

On read hit on page $x$
24: if ($x{\rightarrow}accessed$)
25:     moveToMRUPosition($x$)
26: endif
27: if ($x{\rightarrow}tag$)
28:     prefetch $[x{\rightarrow}Laddr + 1, x{\rightarrow}Laddr + last(x){\rightarrow}p]$
29:     $x{\rightarrow}tag = 0$
30: endif
31: if ($isLast(x)$ && !$x{\rightarrow}old$)
32:     if ($y$ = lastInSequence($x$))
33:         $y{\rightarrow}p = y{\rightarrow}p + readsize$
34:     endif
35: endif

On reading page $x$ (after hit or miss)
36: $x{\rightarrow}accessed = 1$

ON DISK READ COMPLETION:

When a read completes for a set ($s$) of pages
37: createPages($s$, &$last$, &$prev$)
38: $last{\rightarrow}p = (prev\ ?\ prev{\rightarrow}p : 0) + readsize$
39: if ($last{\rightarrow}p \geq APT$)
40:     $last{\rightarrow}g = APT/2$
41:     $(lookup(last{\rightarrow}addr - APT/2)){\rightarrow}tag = 1$
42: endif

When prefetch completes for a set ($s$) of pages
43: createPages($s$, &$last$, &$prev$)
44: $last{\rightarrow}p$ = max($prev{\rightarrow}p$, $last{\rightarrow}g + 1$)
45: $last{\rightarrow}g = prev{\rightarrow}g$
46: if (readWaiting())
47:     $last{\rightarrow}g = last{\rightarrow}g + readsize$
48: endif
49: $(lookup(last{\rightarrow}addr - prev{\rightarrow}g)){\rightarrow}tag = 1$

EVICTION ALGORITHM:

When page $x$ reaches the LRU end
50: if ($x{\rightarrow}old$ || $x{\rightarrow}accessed$)
51:     evict page $x$ from cache
52: else
53:     $x{\rightarrow}old = 1$
54:     moveToMRUPosition($x$)
55:     if ($y$ = lastInSequence($x$))
56:         $y{\rightarrow}p = y{\rightarrow}p - 1$
57:         $y{\rightarrow}g$ = min($y{\rightarrow}g - 1$, $y{\rightarrow}p - 1$)
58:     endif
59: endif
```

Fig. 4. AMP: Main algorithm

IV. Workloads

A. Which workloads to use

The study of caching algorithms using traces is popular as it simplifies the experimental setup while allowing to simulate a real-life workload scenario. Another approach is to use synthetic workload generators which are flexible and can simulate a large number of scenarios for which it is not practical to obtain real-life traces. When the algorithms being tested involve prefetching, using traces is not a good choice. Firstly, the timing of the I/Os is crucial in the context of prefetching algorithms. A read can be a miss or hit depending on the amount of time that passed between consecutive requests. This is not the case with pure demand-paging algorithms where we will get the same hit ratio independent of the timing between the read requests. To ensure that we are faithful to the timing information in the traces we need to run the trace preferably on the same hardware that generated it in the first place. For example, it is impossible to run a trace from a data server with hundreds of disks on a setup with only a few disks. This requires us to either simulate the original hardware on which the trace was collected, or scale the speed of the trace based on the disparity of the two systems. When using older traces, we may also need to factor in the improvement of disk access times. Therefore, using traces for comparison of prefetching algorithms is extremely difficult and an approximation at best. We favor using versatile workload generators which can simulate both simple workloads and complex workloads like OLTP and Video-on-Demand.

B. Sequential Streams

This workload comprises of a continuous series of read requests on consecutive pages with a specified time ($thinktime$) between the requests. Each request is for the specified number of pages ($readsize$). We examine both single stream and multiple stream cases.

C. Short Sequences

Each stream of this kind generates read requests for consecutive pages for the specified sequence length. Once the sequence length is read, the stream randomly selects a new location to start reading another short sequence with the specified $thinktime$ and $readsize$.

D. SPC1-Read workload

SPC-1 ([36], [17]) is a widely used commercial benchmark provided by the Storage Performance Council. It uses a sophisticated workload that simulates business critical environments like OLTP systems, database systems and mail server applications. We use a prototype implementation of the SPC-1 benchmark called SPC1-Read, that matches the specifications of the read component of the SPC1 workload([37]), which comprises uniform random (10%), hierarchical reuse random (65%), and incremental sequential (25%) access patterns. The $thinktime$ and $readsize$ of the constituent workloads are continuously varied according to the probability distribution prescribed by the benchmark. The impact that the write component has on concurrent reads depends on the choice of the size of the write cache, the order of destages as well as the timing of the destages([38]). It serves us well to ignore the write component of the workload so as to clearly appreciate the relative performance of the prefetching algorithms without diluting the results with writes.

E. SPC2-VOD workload

The SPC-2 benchmark is designed to demonstrate the performance of a storage subsystem when running business critical applications that require the large-scale, sequential movement of data. It is comprised of tests that simulate applications characterized by large I/Os. One such test which is purely reads (and hence of interest to us), is the *Video on Demand* test which simulates individualized video entertainment provided to a community of subscribers, by drawing from a digital film library [39]. The workload creates the specified number of sequential streams that read 256 KB on each I/O with a thinktime of 333.3 ms.

V. Experimental Set-up

A. The Basic Hardware Set-up

We use an IBM xSeries 345 machine equipped with two Intel Xeon 2 GHz processors, 4 GB DDR, and six 10K RPM SCSI disks (IBM, 06P5759, U160) of 36.4 GB each. A Linux kernel (version 2.6.11) runs on this machine hosting all our workload generators and cache simulation framework. We employ five SCSI disks for the purposes of our experiments, and the remaining one for the operating system, our software, and workloads.

B. Software Setup

We implemented a framework, called *WorkGen* as shown in Figure 5, which allows us to benchmark various algorithms across a large range of cache sizes, varying page sizes, and consumption rates. *WorkGen* is divided into three layers. The first layer consists of the workload modules, which can simulate any number of parallel streams consisting of sequential, short sequential, SPC1-Read, or SPC2-VOD workloads. The second layer consists of the prefetching modules, which allows us to select any of the prefetching algorithms that we implemented. Each algorithm module has access to the LRU data structure as well as the read(), and prefetch() methods used to perform demand reads and prefetches respectively. The third layer consists of the disk backend which is the target for all the I/Os.

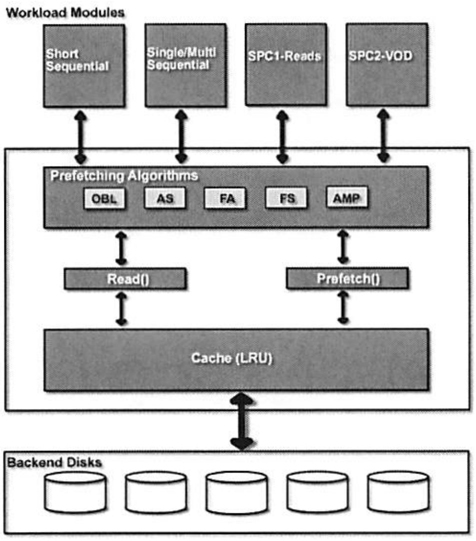


Fig. 5. WorkGen Architecture

C. The Competitors

We implemented prefetching algorithms that represent the FS, FA, and AS classes.

Within the FS and FA classes we pick members with small, medium and large p. In Figure 6, we observe that for the FA algorithms there is no optimal fixed value for g that works for all workloads. We have chosen g to be half of p as that works best for the widest variety of workloads. For AS algorithms, we chose two popular variants, which adapt p linearly ($\text{AS}_{\text{Linear}}$) and exponentially (AS_{Exp}). We also compare with OBL and the case with no prefetching.

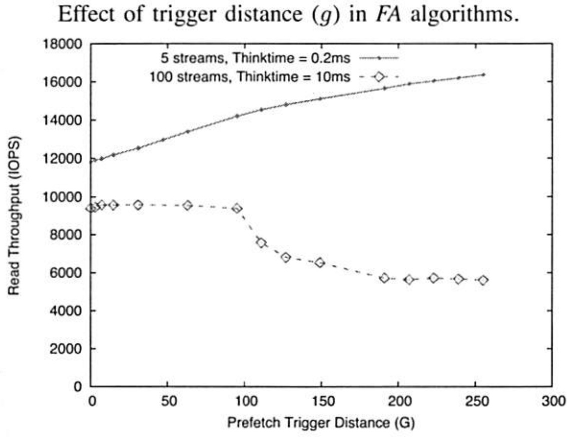


Fig. 6. On x-axis we vary the trigger distance (g) in an FA algorithm with $p = 256$. On the y-axis we show the throughput when using a 120 MB cache. When using five fast streams we get higher throughput with higher values of g, whereas, with a hundred slower streams, a smaller g performs better.

D. Measuring success

The ultimate goal of any cache management algorithm is to improve the shape of the throughput-response time curve for the system by lowering the response times and increasing the throughput across all workloads. Most caching research has focused on minimizing miss ratios (or maximizing hit ratios) which at best is a good heuristic for improving performance of a system. To be fair it is not just the miss ratio but also the average cost of misses that impacts the aggregate response time. For example, an aggressive prefetching algorithm can potentially reduce the miss ratio but suffer a severe increase in the average cost of misses as it overloads the disks. In fact, with prefetching, the concept of a read miss itself is nebulous because a read that happens after a prefetch request for the page has been issued and before the prefetch actually completes is somewhere between a hit and a miss, but technically neither. Even in the absence of prefetching, some disks might be less busy than others leading to smaller miss penalties on those disks. Even on a single disk reading from an area that is not visited often by the disk head tends to be more expensive. In short, it is prudent to measure performance in terms of aggregate read response times and throughput whenever possible.

Another quantity which is useful is the *stall time*. It is the total time for which application had to wait because the requested data was not present in the cache. This is very closely related to the aggregate throughput as a lower stall time results in correspondingly higher throughput. We however choose to report in terms of throughput as it is more immediately relevant to performance.

VI. Results

A. Single Sequential Stream

Our goal is to create an intimate understanding of the behavior of various sequential prefetching algorithms. We implemented 9 prefetching algorithms and compared them with AMP. In Figure 7, we examine the actual throughput achieved as a function of the requested rate by a single sequential stream when assisted by various prefetching algorithms.

As expected, we observe the lowest throughput for no prefetch. The One Block Lookahead (OBL) algorithm performs about the same because we prefetch only one page and the request size is 2 pages. A two or more block lookahead algorithm would have worked better by reducing the amount of misses. This is precisely the intent of the Fixed Synchronous (FS) class of algorithms which have a fixed degree of prefetch (p).

FA algorithms are superior to the FS algorithms because they can start a prefetch on a hit thereby avoiding more misses than the FS algorithms. Both the

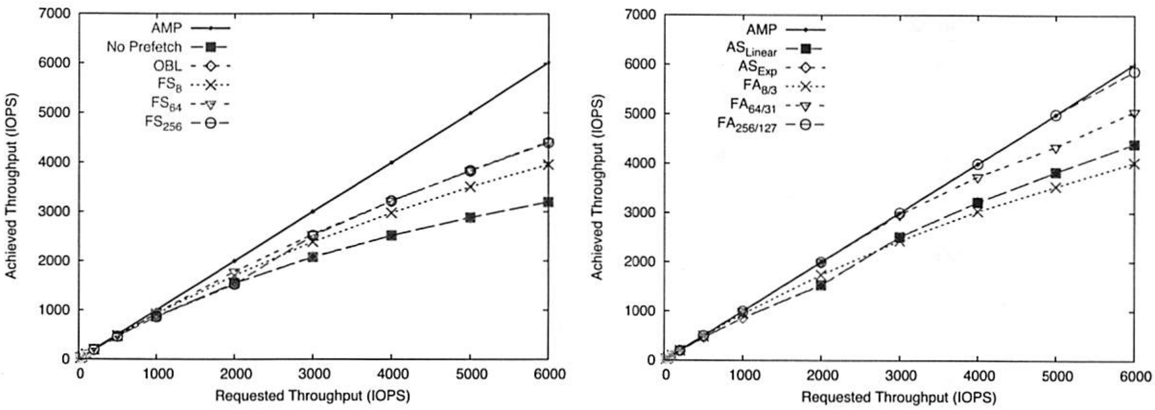

Single Sequential Stream with varying request rates

Fig. 7. We show the achieved throughput as a function of the requested throughput for 8 KB reads using a 100 MB cache and one SCSI disk. For clarity, we split the comparison with AMP into two panels. AMP keeps up with the requested throughput and outperforms all other algorithms.

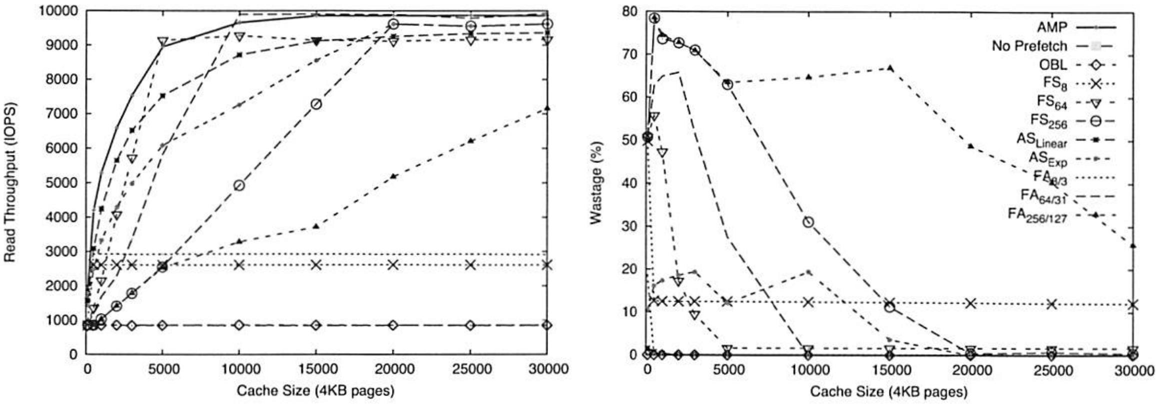

Multiple Sequential Streams with varying cache sizes

Fig. 8. On the left panel, we show the average achieved throughput over a period of two minutes as a function of the cache size for 100 concurrent streams reading from five SCSI disks with thinktime = 10 ms and readsize = 8 KB. On the right panel, we show the corresponding wastage percentage (unaccessed pages evicted / total pages evicted * 100%).

FS and FA algorithm results seem to indicate that the larger the p, the better the performance. This is true for a single sequential stream where the cache is abundantly available. In multiple stream experiments which compete for cache space, we will better appreciate the need for a careful choice of p.

We also measure the performance of the adaptive synchronous algorithms. AS_{Linear} and AS_{Exp} increase p as the detected sequence becomes longer. For all the adaptive algorithms in this paper we uniformly use a maximum degree of prefetch of 256 for a fair comparison. Both AS variants perform roughly the same as they adapt and reach this maximum value of p during the first few seconds on the experiment.

AMP being able to adapt not only p but also g convincingly outperforms the FA algorithms by 2-50%, the AS algorithms by 37%, the FS algorithms by 37-52% and no prefetching and OBL by 88%. AMP closely followed by $FA_{256/127}$ was able to satisfy the request rate throughout the single stream experiment.

B. Multiple Sequential Streams: varying cache size

It is extremely important to examine the common case where a limited cache is used by prefetching algorithms to cater to multiple sequential streams. In Figure 8, we depict the aggregate throughput achieved by a hundred parallel sequential streams with a thinktime of 10 ms. The key observation is that different algorithms are suitable for different cache sizes, while AMP is universally the best, closely enveloping all the other plots.

Out of the FS algorithms, FS_8 is the best at very low cache sizes, while FS_{64} is much better at the higher cache sizes. In FA algorithms, $FA_{8/3}$ is again the best

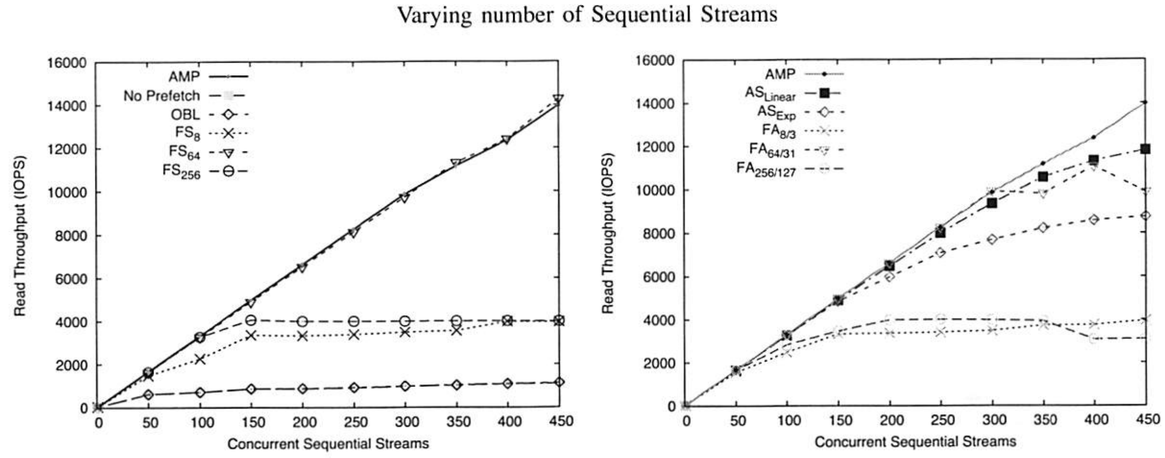


Fig. 9. We show the average achieved throughput over a period of two minutes as a function of the number of concurrent sequential streams reading from five SCSI disks with thinktime = 30 ms, readsize = 8 KB, and a 100 MB cache.

at low cache sizes, while $\mathsf{FA}_{256/127}$ is much superior at the higher cache sizes. As a rule of thumb, a lower prefetch degree performs better at lower cache sizes as high values of p create prefetch wastage in smaller caches. This is in contrast to the single stream case, where a higher p was always a better choice. This leads us to the adaptive synchronous algorithms. $\mathsf{AS}_{\mathrm{Exp}}$ increases p exponentially, reaching higher values of p quickly creating significant prefetch wastage for lower cache sizes. $\mathsf{AS}_{\mathrm{Linear}}$, being slower in its adaptation, usually performs better than $\mathsf{AS}_{\mathrm{Exp}}$ at lower cache sizes while the reverse is true at higher cache sizes. The only algorithm that truly adapts the value of p so as to minimize prefetch wastage is AMP. As a rough measure of the overall performance of each algorithm, we compute the average throughput across all cache sizes in Figure 8. We find that AMP outperforms the FA algorithms by 29-172%, the AS algorithms by 12-24%, the FS algorithms by 21-210% and no prefetching and OBL by a factor of 8. This is also a testimonial to the fact that AMP algorithm closely follows the optimality criteria derived in Section III-B.

In the right panel of Figure 8 we plot wastage, which is the percentage of evicted pages that were evicted before they could be accessed. We observe that the prefetching algorithms that have a high p or aggressively increase p suffer from the most prefetch wastage, and that wastage is larger for smaller cache sizes. The prefetch wastage in the case of AMP is always less than 0.1%, while on an average, FA, FS, and AS algorithms waste up to 60%, 41%, and 11% respectively.

C. Multiple Sequential Streams: varying number of streams

In Figure 9, we study the aggregate streaming throughput of various prefetching algorithms when we increase the number of concurrent sequential streams while keeping the cache size constant. We observe that most algorithms saturate at some throughput beyond which increasing the number of streams does not improve the aggregate throughput. Algorithms that issue fewer but larger disk reads and at the same time waste little generally do better. We observe that no prefetching and OBL have the lowest throughput as they have a large number of small read requests. Somewhat better are the $\mathsf{FA}_{8/3}$ and FS_8 algorithms as they create fewer read requests than OBL. Interestingly, $\mathsf{FA}_{256/127}$ and FS_{256} algorithms also have similarly low performance in spite of large prefetch degree. This is because the large prefetch degree leads to significant prefetch wastage. The FS_{64} and $\mathsf{FA}_{64/31}$ perform the best in their respective classes as they strike a balance and have large reads but do not waste as much. The $\mathsf{AS}_{\mathrm{Linear}}$ and $\mathsf{AS}_{\mathrm{Exp}}$ are generally good performers because they adapt the degree the prefetch. However, since these algorithms lack the ability to detect and avoid wastage, the more aggressive $\mathsf{AS}_{\mathrm{Exp}}$ fares worse than its linear counterpart. AMP being an asynchronous adaptive algorithm discovers the right prefetch degree for each stream thus avoiding wastage and achieving the best possible performance.

At the maximum number of streams, AMP outperforms the FA algorithms by 41-350%, the AS algorithms by 18-60%, the FS algorithms from nearly equal to 252% and no prefetching and OBL by a factor of 12.

D. SPC1-Read workload

We study the impact of the various prefetching algorithms on the performance of the cache when subjected to the read component of the SPC1 benchmark workload. In Figures 10, 11 we show the aggregate response time as a function of the obtained throughput. Lower plots indicate better performance as at the same

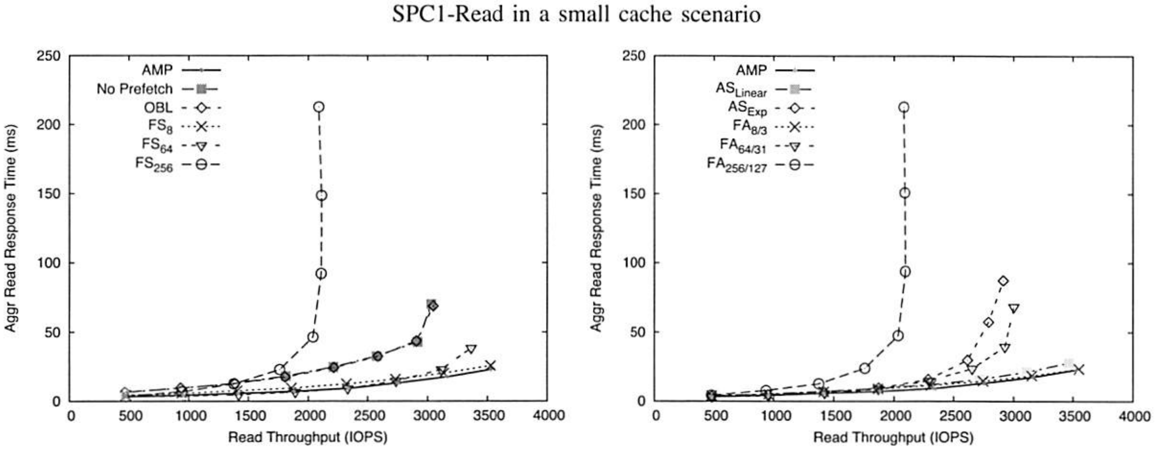


Fig. 10. We measure the aggregate read response time as a function of the achieved throughput when running the read portion of the SPC1 benchmark in a small cache scenario: 120 MB cache, backend=3.5 GB. The above graph also shows an example where no prefetching performs better than aggressive prefetching algorithms like FS_{256} , $\mathsf{FA}_{256/127}$ and $\mathsf{AS}_{\mathrm{Exp}}$. This underscores the importance of wisely adapting p according to the workload.

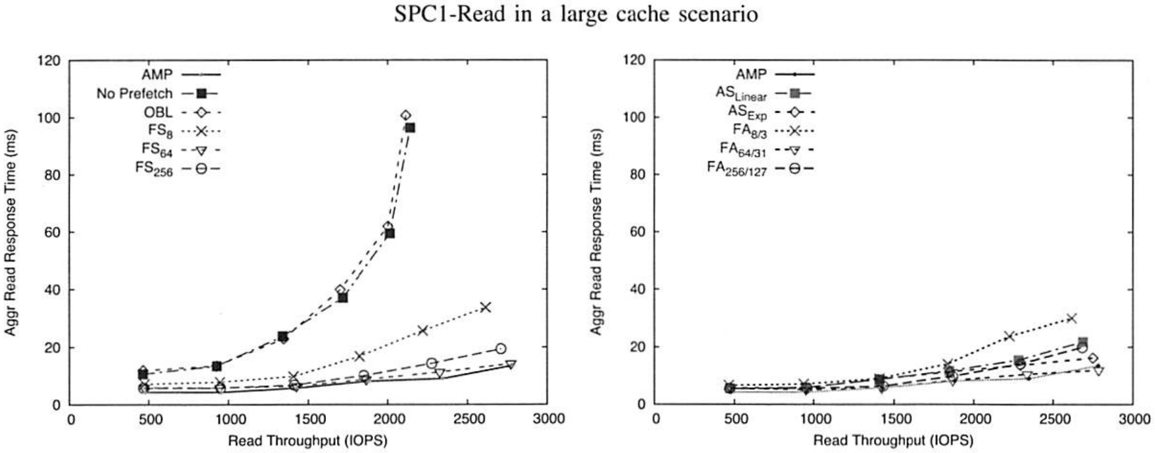


Fig. 11. We measure the aggregate read response time as a function of the achieved throughput when running the read portion of the SPC1 benchmark in a large cache scenario: 2 GB cache, backend = 35 GB

response time a higher throughput can be achieved. Clearly, AMP is consistently the best performing algorithm in both the small cache and large cache scenarios. No other algorithm can perform well in both scenarios. When the cache is large, the algorithms with a higher prefetch degree seem to do better, while in the small cache scenario, where the cache is precious, the algorithms that are conservative in their prefetch degree tend to perform much better as they incur lesser prefetch wastage. AMP is able to quickly adapt to different workloads and cache sizes, hence performing the best among all the algorithms.

E. Short Sequences

In Section III-B, we derived the optimality criteria for sequential prefetching in the steady state. We have not discussed the behavior of the sequential prefetching algorithms when the average length of sequences is rather short. Apart from providing close to optimal performance for long streams, AMP achieves the best overall performance for short streams as well. Figure 12 shows the throughput of various algorithms as the length of sequences go from 1 (effectively random) to 8192 read I/Os. The AS algorithms along with FS_8 and $\mathsf{FA}_{8/3}$ perform well for short sequence lengths as they have a smaller p and suffer from less prefetch wastage. As the sequence lengths are increased, the $\mathsf{FA}_{256/127}$ becomes a strong contender. The fact that AMP starts off with a small p and adapts to make it larger if necessary makes it perform reasonably well for short sequences. As the length of sequences becomes larger, the adaptive power of AMP allows it discover the right combination of p and g. AMP is therefore not only provably optimal for steady sequential streams but also has the best overall performance for short sequences as well.

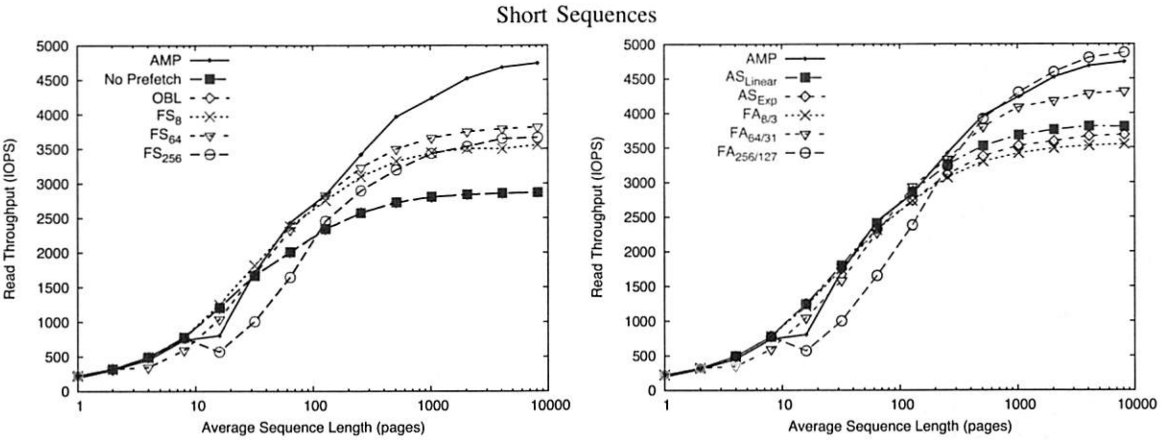


Fig. 12. We show the achieved throughput as a function of the average sequence length ranging from 1 I/O (essentially random) to 8192 I/Os. We use a single stream with thinktime = 0.2 ms and a 100 MB cache.

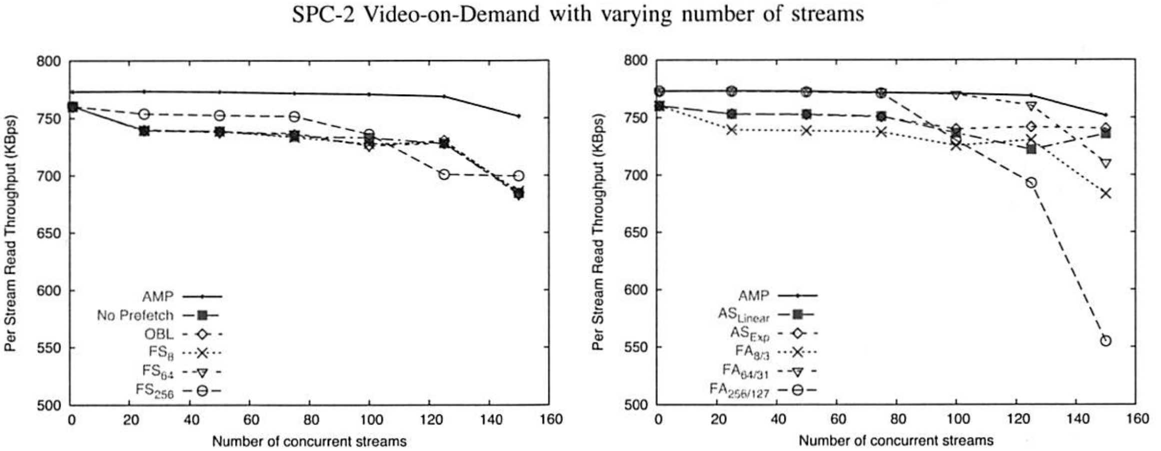


Fig. 13. We measure the achieved throughput per stream as the number of concurrent video-on-demand streams increases using five SCSI disks and a cache of 100 MB. The SPC2-VOD workload uses read size of 256 KB, thinktime = 333.3 ms.

F. SPC2-VOD workload

In Figure 13 we measure the performance for video-on-demand workloads. The goal is to provide each sequential stream its requisite bandwidth (768 KBps in the case of SPC2-VOD workload) for the maximum number of streams. We can easily observe that AMP is able to entertain the most number of concurrent streams (up to 125) at the desired bandwidth. $FA_{64/31}$ starts failing at about 100 streams and $FA_{256/127}$ fails after 75 streams because of more severe prefetch wastage. None of the other algorithms can match the demanded rate as they incur expensive read misses which stall the client and lower the throughput.

VII. Conclusions

Sequential prefetching is the most widely used prefetching technique in storage subsystems. We have argued the need for an algorithm that can adapt both the prefetch degree and the trigger distance on a per stream basis in response to evolving workloads. We have provided a theoretical analysis and proved the sufficient conditions for optimal online cache management for steady-state sequential streams. We also presented a novel, simple, adaptive and practical implementation called AMP. We have demonstrated through a series of wide ranging experiments including realistic benchmarks, that AMP provides the highest possible aggregate throughput when a cache is shared among multiple sequential streams. Even in scenarios where the sequential streams are not steady, comprise of short sequences, or are intermixed with random workloads (as in SPC1-Read), we demonstrated that AMP convincingly outperforms all competing algorithms by wasting the least amount of cache while providing the best overall throughput.

We anticipate AMP to be widely applicable not only in storage subsystems, but in any system that services sequential workload.

REFERENCES

[1] A. Rogers and K. Li, "Software support for speculative loads," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, vol. 27(9), (New York, NY), pp. 38–50, ACM Press, 1992.

[2] K. K. David Callahan and A. Porterfield, "Software prefetching," in *ACM SIGARCH Computer Architecture News*, vol. 19(2), (New York, NY), pp. 40–52, ACM Press, 1991.

[3] C. Metcalf, "Data prefetching: a cost/performance analysis," 1993.

[4] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *SOSP*, pp. 79–95, 1995.

[5] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, no. 2, pp. 87–106, 1991.

[6] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *Proceedings 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News*, vol. 18(3), pp. 354–368, 1990.

[7] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, vol. 27(9), (New York, NY), pp. 51–61, ACM Press, 1992.

[8] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Architectural Support for Programming Languages and Operating Systems*, pp. 222–233, 1996.

[9] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 115–126, 1998.

[10] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger, "SPAID: Software prefetching in pointer- and call-intensive environments," in *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 231–236, 1995.

[11] T. M. Kroeger, D. D. E. Long, and J. C. Mogul, "Exploring the bounds of web latency reduction from caching and prefetching," in *USENIX Symposium on Internet Technologies and Systems*, 1997.

[12] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical prefetching via data compression," pp. 257–266, 1993.

[13] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," in *USENIX Summer*, pp. 197–207, 1994.

[14] D. Kotz and C. S. Ellis, "Practical prefetching techniques for parallel file systems," in *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pp. 182–189, IEEE Computer Society Press, 1991.

[15] K. S. Grimsrud, J. K. Archibald, and B. E. Nelson, "Multiple prefetch adaptive disk caching," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 1, pp. 88–103, 1993.

[16] S. Harizopoulos, C. Harizakis, and P. Triantafillou, "Hierarchical caching and prefetching for high performance continuous media servers with smart disks," *IEEE Concurrency*, vol. 8, no. 3, pp. 16–22, 2000.

[17] B. S. Gill and D. S. Modha, "SARC: Sequential prefetching in adaptive replacement cache," in *Proceedings of the USENIX 2005 Annual Technical Conference*, pp. 293–308, 2005.

[18] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[19] F. Dahlgren, M. Dubois, and P. Stenström, "Fixed and adaptive sequential prefetching in shared memory multiprocessors.," in *ICPP*, pp. 56–63, 1993.

[20] M. K. Tcheun, H. Yoon, and S. R. Maeng, "An adaptive sequential prefetching scheme in shared-memory multiprocessors.," in *ICPP*, pp. 306–313, 1997.

[21] T. Cortes and J. Labarta, "Linear aggressive prefetching: A way to increase the performance of cooperative caches," in *Proceedings of the Joint International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing*, (San Juan, Puerto Rico), pp. 45–54, 1999.

[22] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proceedings of the 18th annual international symposium on computer architecture*, (Toronto, Intario, Canada), pp. 54–63, 1991.

[23] R. L. Lee, P.-C. Yew, and D. H. Lawrie, "Data prefetching in shared memory multiprocessors.," in *ICPP*, pp. 28–31, 1987.

[24] T.-F. Chen and J.-L. Baer, "Effective hardware based data prefetching for high-performance processors.," *IEEE Trans. Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[25] F. Dahlgren and P. Stenström, "Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors:," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 4, pp. 385–398, 1996.

[26] K. S. Grimsrud, J. K. Archibald, and B. E. Nelson, "Multiple prefetch adaptive disk caching.," *IEEE Trans. Knowl. Data Eng.*, vol. 5, no. 1, pp. 88–103, 1993.

[27] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, 1999.

[28] J. S. Vitter and P. Krishnan, "Optimal prefetching via data compression," *Journal of the ACM*, vol. 43, no. 5, pp. 771–793, 1996.

[29] H. Lei and D. Duchamp, "An analytical approach to file prefetching," in *1997 USENIX Annual Technical Conference*, (Anaheim, California, USA), 1997.

[30] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," in *Measurement and Modeling of Computer Systems*, pp. 188–197, 1995.

[31] M. Kallahalla and P. J. Varman, "Pc-opt: Optimal offline prefetching and caching for parallel i/o systems.," *IEEE Trans. Computers*, vol. 51, no. 11, pp. 1333–1344, 2002.

[32] T. Kimbrel and A. R. Karlin, "Near-optimal parallel prefetching and caching," in *IEEE Symposium on Foundations of Computer Science*, pp. 540–549, 1996.

[33] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. R. Karlin, and K. Li, "A trace-driven comparison of algorithms for parallel prefetching and caching," in *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pp. 19–34, USENIX Association, 1996.

[34] P. Reungsang, S. K. Park, S.-W. Jeong, H.-L. Roh, and G. Lee, "Reducing cache pollution of prefetching in a small data cache.," in *ICCD*, pp. 530–533, 2001.

[35] P. Jain, S. Devadas, and L. Rudolph, "Controlling cache pollution in prefetching with software-assisted cache replacement," Tech. Rep. CSG-462, M.I.T., 2001.

[36] B. McNutt and S. Johnson, "A standard test of I/O cache," in *Proc. Comput. Measurements Group's 2001 Int. Conf.*, 2001.

[37] Storage Performance Council, "SPC Benchmark-1: Specification, version 1.10.1," September 2006.

[38] B. S. Gill and D. S. Modha, "WOW: Wide ordering of writes - combining spatial and temporal locality in non-volatile caches," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pp. 129–142, 2005.

[39] Storage Performance Council, "SPC Benchmark-2: Specification, version 1.2," September 2006.

Nache: Design and Implementation of a Caching Proxy for NFSv4

Ajay Gulati
Rice University
gulati@rice.edu

Manoj Naik
IBM Almaden Research Center
manoj@almaden.ibm.com

Renu Tewari
IBM Almaden Research Center
tewarir@us.ibm.com

Abstract

In this paper, we present Nache, a caching proxy for NFSv4 that enables a consistent cache of a remote NFS server to be maintained and shared across multiple local NFS clients. Nache leverages the features of NFSv4 to improve the performance of file accesses in a wide-area distributed setting by bringing the data closer to the client. Conceptually, Nache acts as an NFS server to the local clients and as an NFS client to the remote server. To provide cache consistency, Nache exploits the read and write delegations support in NFSv4. Nache enables the cache and the delegation to be shared among a set of local clients, thereby reducing conflicts and improving performance. We have implemented Nache in the Linux 2.6 kernel. Using Filebench workloads and other benchmarks, we present the evaluation of Nache and show that it can reduce the NFS operations at the server by 10-50%.

1 Introduction

Most medium to large enterprises store their unstructured data in filesystems spread across multiple file servers. With ever increasing network bandwidths, enterprises are moving toward distributed operations where sharing presentations and documents across office locations, multi-site collaborations and joint product development have become increasingly common. This requires sharing data in a uniform, secure, and consistent manner across the global enterprise with reasonably good performance.

Data and file sharing has long been achieved through traditional file transfer mechanisms such as FTP or distributed file sharing protocols such as NFS and CIFS. While the former are mostly adhoc, the latter tend to be "chatty" with multiple round-trips across the network for every access. Both NFS and CIFS were originally designed to work for a local area network involving low latency and high bandwidth access among the servers and clients and as such are not optimized for access over a wide area network. Other filesystem architectures such as AFS [17] and DCE/DFS[20] have attempted to solve the WAN file sharing problem through a distributed architecture that provides a shared namespace by uniting disparate file servers at remote locations into a single logical filesystem. However, these technologies with proprietary clients and protocols incur substantial deployment expense and have not been widely adopted for enterprise-wide file sharing. In more controlled environments, data sharing can also be facilitated by a clustered filesystem such as GPFS[31] or Lustre[1]. While these are designed for high performance and strong consistency, they are either expensive or difficult to deploy and administer or both.

Recently, a new market has emerged to primarily serve the file access requirements of enterprises with outsourced partnerships where knowledge workers are expected to interact across a number of locations across a WAN. Wide Area File Services (WAFS) is fast gaining momentum and recognition with leading storage and networking vendors integrating WAFS solutions into new product offerings[5][3]. File access is provided through standard NFS or CIFS protocols with no modifications required to the clients or the server. In order to compensate for high latency of WAN accesses, low bandwidth and lossy links, the WAFS offerings rely on custom devices at both the client and server with a custom protocol optimized for WAN access in between.

One approach often used to reduce WAN latency is to cache the data closer to the client. Another is to use a WAN-friendly access protocol. The version 4 of the NFS protocol added a number of features to make it more suitable for WAN access [29]. These include: batching multiple operations in a single RPC call to the server, enabling read and write file delegations for reducing cache consistency checks, and support for redirecting clients to other, possibly closer, servers. In this paper, we discuss the design and implementation of a caching file server proxy called Nache. Nache leverages the features of NFSv4 to improve the performance of file serving in a wide-area distributed setting. Basically, the Nache proxy sits in between a local NFS client and a remote NFS server bringing the remote data closer to the client. Nache acts as an NFS server to the local client and as an NFS client to the remote server. To provide cache consistency, Nache exploits the read and write delegations support in NFSv4. Nache is ideally suited for environments where data is commonly shared across multiple clients. It provides a consistent view of the data by allowing multiple clients to share a delegation, thereby removing the overhead of a recall on a conflicting access. Sharing files across clients is common for read-only data front-ended by web servers, and is becoming widespread for presentations, videos, documents and collaborative projects across a distributed enterprise. Nache is beneficial even when the degree of sharing is small as it reduces both the response time of a WAN access and the overhead of recalls.

In this paper, we highlight our three main contributions. First, we explore the performance implications of read and write open delegations in NFSv4. Second, we detail the implementation of an NFSv4 proxy cache architecture in the Linux 2.6 kernel. Finally, we discuss how delegations are leveraged to provide consistent caching in the proxy. Using our testbed infrastructure, we demonstrate the performance benefits of Nache using the *Filebench* benchmark and other workloads. For these workloads, the Nache is shown to reduce the number of NFS operations seen at the server by 10-50%.

The rest of the paper is organized as follows. In the next section we provide a brief background of consistency support in various distributed filesystems. Section 3 analyzes the delegation overhead and benefits in NFSv4. Section 4 provides an overview of the Nache architecture. Its implementation is detailed in section 5 and evaluated in section 6 using different workloads. We discuss related work in section 7. Finally, section 8 presents our conclusions and describes future work.

2 Background: Cache Consistency

An important consideration in the design of any caching solution is cache consistency. Consistency models in caching systems have been studied in depth in various large distributed systems and databases [9]. In this section, we review the consistency characteristics of some widely deployed distributed filesystems.

- *Network File System (NFS):* Since perfect coherency among NFS clients is expensive to achieve, the NFS protocol falls back to a weaker model known as *close-to-open* consistency [12]. In this model, the NFS client checks for file existence and permissions on every open by sending a GETATTR or ACCESS operation to the server. If the file attributes are the same as those just after the previous close, the client will assume its data cache is still valid; otherwise, the cache is purged. On every close, the client writes back any pending changes to the file so that the changes are visible on the next open. NFSv3 introduced weak cache consistency [28] which provides a way, albeit imperfect, of checking a file's attributes before and after an operation to allow a client to identify changes that could have been made by other clients. NFS, however, never implemented distributed cache coherency or concurrent write management [29] to differentiate between updates from one client and those from multiple clients. Spritely NFS [35] and NQNFS [23] added stronger consistency semantics to NFS by adding server callbacks and leases but these never made it to the official protocol.
- *Andrew File System (AFS):* Compared to NFS, AFS is better suited for WAN accesses as it relies heavily on client-side caching for performance [18]. An AFS client does whole file caching (or large chunks in later versions). Unlike an NFS client that checks with the server on a file open, in AFS, the server establishes a callback to notify the client of other updates that may happen to the file. The changes made to a file are made visible at the server when the client closes the file. When there is a conflict with a concurrent close done by multiple clients, the file at the server reflects the data of the last client's close. DCE/DFS improves upon AFS caching by letting the client specify the type of file access (read, write) so that the callback is issued only when there is an open mode conflict.
- *Common Internet File System (CIFS):* CIFS enables stronger cache consistency [34] by using opportunistic locks (OpLocks) as a mechanism for cache coherence. The CIFS protocol allows a client to request three types of OpLocks at the time of file open: exclusive, batch and level II. An exclusive oplock enables it to do all file operations on the file without any communication with the server till the file is closed. In case another client requests access to the same file, the oplock is revoked and the client is required to flush all modified data back to the server. A batch oplock permits the client to hold on to the oplock even after a close if it plans to reopen the file very soon. Level II oplocks are shared and are used for read-only data caching where multiple clients can simultaneously read the locally cached version of the file.

3 Delegations and Caching in NFSv4

The design of the version 4 of the NFS protocol [29] includes a number of features to improve performance in a wide area network with high latency and low bandwidth links. Some of these new features are:

1. *COMPOUND RPC:* This enables a number of traditional NFS operations (LOOKUP, OPEN, READ, etc.) to be combined in a single RPC call to the server to carry out a complex operation in one network round-trip. COMPOUND RPCs provide lower overall network traffic and per command round trip delays.
2. *Client redirection:* The NFSv4 protocol provides a special return code (`NFS4ERR_MOVED`) and a filesystem attribute (`fs_locations`) to allow an NFS client to be redirected to another server at filesystem boundaries. Redirection can be used for building a wide-area distributed federation of file servers with a common namespace where data can be replicated and migrated among the various file servers.
3. *OPEN delegations:* File delegation support in NFSv4 is a performance optimization which eliminates the need for the client to periodically check with the server for cache consistency. Later in this section, we will investigate delegations in detail as they form an important component of Nache.

By granting a file delegation, the server voluntarily cedes control of operations on the file to a client for the duration of the client lease or until the delegation is recalled. When a file is delegated, all file access and modification requests can be handled locally by the client without sending any network requests to the server. Moreover, the client need not periodically validate the cache as is typically done in NFS as the server guarantees that there will be no other conflicting access to the file. In fact, the client need not flush modified data on a CLOSE as long as the server guarantees that it will have sufficient space to accept the WRITEs when they are done at a later time.

NFSv4 delegations are similar to CIFS oplocks but are not exactly the same [29]. Delegations in NFSv4 are purely a server driven optimization, and without them, the standard client-side caching rules apply. In CIFS, on the other hand, oplocks are requested by the client and are necessary for caching and coherency. If oplocks are not available, a CIFS client cannot use its cached data and has to send all operations to the server. In addition, NFSv4 delegations can be retained at the clients across file CLOSE as in CIFS batch oplocks.

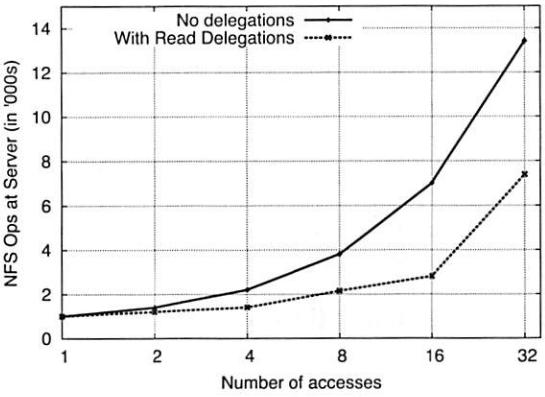


Figure 1: *Performance of Read Delegations: The graph shows the number of NFS ops (packets) sent to the server with and without read delegations. The X axis denotes the number of times a single NFSv4 client opens, reads and closes a file.*

3.1 Read Delegation

A read delegation is awarded by the server to a client on a file OPENed for reading (that does not deny read access to others). The decision to award a delegation is made by the server based on a set of conditions that take into account the recent history of the file [32]. In the Linux NFSv4 server, for example, the read delegation is awarded on the second OPEN by the same client (either after opening the same file or on opening another file) [13]. After the delegation is awarded, all READs can be handled locally without sending GETATTRs to check cache validity. As long as the delegation is active, OPEN, CLOSE and READ requests can be handled locally. All LOCK requests (including non exclusive ones) are still sent to the server. The delegation remains in effect for the lease duration and continues when the lease is renewed. Multiple read delegations to different clients can be outstanding at any time. A callback path is required before a delegation is awarded so that the server can use it to recall a delegation on a conflicting access to a file such as an OPEN for write, RENAME, and REMOVE. After a delegation has been recalled, the client falls back to traditional attribute checking before reading cached data.

Currently, the NFSv4 server hands out delegations at the granularity of a file. Directory delegations are being considered in future revisions of the protocol [21], but the design of Nache relies only on file level delegations.

To illustrate the benefit of read delegations, both in terms of server response time and message overhead, we measured the values for a single NFSv4 client, with multiple application processes, iterating over a OPEN-READ-CLOSE operation sequence on a file. Both the client and server were running Linux kernel version 2.6.17 [13]. Figure 1 shows the number of NFS operations processed by the server with and without delegations. As shown, the server load in terms of number of packets received is reduced by 50% with read delegations enabled. Further gains are achieved when the file is cached for a long time by saving on the additional GETATTRs sent to validate the cache after a timeout. We observe that if the file is cached beyond the attribute timeout (typically 30 seconds), the operations at the server with delegations reduced by another 8%.

3.2 Write Delegation

Similar to read delegations, write delegations are awarded by the server when a client opens a file for write (or read/write) access. While the delegation is outstanding, all OPEN, READ, WRITE, CLOSE, LOCK, GETATTR, SETATTR requests for the file can be handled locally by the client. Handling delegations for write, however, is far more complicated than that for reads. The server not only checks if a callback path exists to the client to revoke the delegation, it also limits the maximum size that the client can write to prevent ENOSPC errors since client has the option to flush data lazily. On a conflicting OPEN by another client, the server recalls the delegation which triggers the client to commit all dirty data and return the delegation. The conflicting OPEN is delayed until the delegation recall is complete. The failure semantics with write delegations are also complicated, given that the client can have dirty data that has not yet been committed at the server even after a CLOSE. To maintain the close-to-open cache consistency semantics, the client, even with a write delegation, may flush all dirty data back to the server on a CLOSE. Observe that, with write delegations, the consistency semantics are slightly tighter than a pure close-to-open model. The second client, on an OPEN after a delegation recall, sees the data written by the first client before the first client closes the file.

At the time of writing, the Linux server implementation (2.6.17) only hands out read delegations. We have a prototype implementation to enable write delegations which we discuss in detail in section 6. Figure 2 shows the NFS operations processed at the server with and without write delegations. The workload is the same as in the previous experiment with a single NFSv4 client iterating over a sequence of OPEN-WRITE/READ-CLOSE operations on a file with a balanced number of reads and writes. The server load, in terms of number of packets received, is reduced by 5 times with write delegations enabled. One would expect write delegations to provide significantly better performance than what is observed. The reason this is not the case is that write delegations are not completely implemented in our prototype. Although the delegation is granted, a client still sends WRITE requests to the server, whereas they need not once a delegation is obtained.

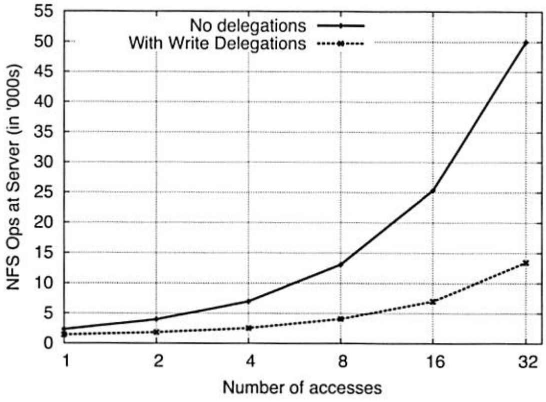


Figure 2: *Performance of Write Delegations: The graphs shows the number of ops (NFS packets) sent to the server with and without write delegations. The X axis denotes the number of times a single NFSv4 client opens, writes/reads and closes a file.*

As we have demonstrated, delegations can substantially improve cache performance and coherency guarantees for a single client that is frequently accessing a file. However, if read-write sharing is common (as is the case in joint software development), write delegations, if not granted intelligently, may make matters worse.

	LAN access	WAN access
No delegations	1.5-9 ms	150-500ms
Read delegation	1007 ms	1400 ms
Write delegation	1010 ms	1600 ms

Table 1: *Overhead of recalling a delegation. The table shows the time taken to complete a conflicting OPEN with delegation recall over a LAN and a WAN.*

Table 1 measures the overhead of recalling a delegation in terms of the delay observed by a conflicting OPEN (with read/write access) for both read and write delegations. The second OPEN is delayed until the delegation is recalled from the first client. With write delegations this also includes the time taken to flush all dirty data back to the server. We observe that the recall of a read delegation adds a one second delay to the second OPEN on Linux. This is because the NFSv4 client waits one second before retrying the OPEN on receiving a delay error from the server. The overhead with write delegations was similar to that with read delegations as the overhead of flushing data is low for a file smaller than 2MB. However, for large file writes the recall time is substantial. Figure 3 shows the client OPEN time with a write delegation recall when the size of the dirty data varies from 256KB to 32MB.

Clearly, the above experiments show that delegations

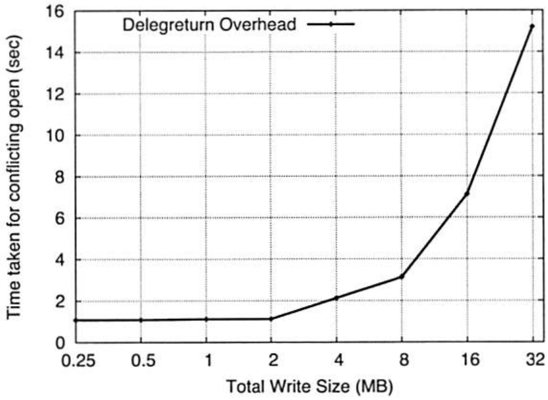


Figure 3: *Overhead of recalling a write delegation. The X-axis is the size of the data written to the file. The Y-axis is the time taken to complete a conflicting OPEN.*

are beneficial if a single client is exclusively accessing a file; and that any conflict can substantially affect the client response time. However, if it was possible for clients to *share* the delegations there would be no conflict. The underlying principle of Nache is to extend delegations and caching from a single client to a set of local clients that share both the cache and the delegation, thereby minimizing conflict. Even with read delegations that can be simultaneously awarded, Nache adds the benefit of a shared cache.

4 Nache Overview

The need for the Nache caching proxy arose from our earlier work on building a wide-area federated filesystem leveraging NFSv4 [16]. The goal of this system was to provide a uniform filesystem view across heterogeneous file servers interconnected by enterprise-wide or Internet-scale networks. Figure 4 shows the distribution of file servers and the common namespace view that they all export to the clients. The namespace was created by leveraging the client redirection feature of NFSv4. The client can mount the root of the common namespace tree from any server in the federation and see a uniform namespace.

4.1 Caching Proxy vs. Redirection

In such a federated system, with data on different geographically distributed physical locations, there are two models for data access: *client redirection and data shipping*. In the client redirection model, the client is referred to the server where the data actually resides for fetching it over the network. In the data shipping model, data from the remote server is cached on a server closer to the client to reduce frequent WAN accesses. Figure 5 shows the two access models.

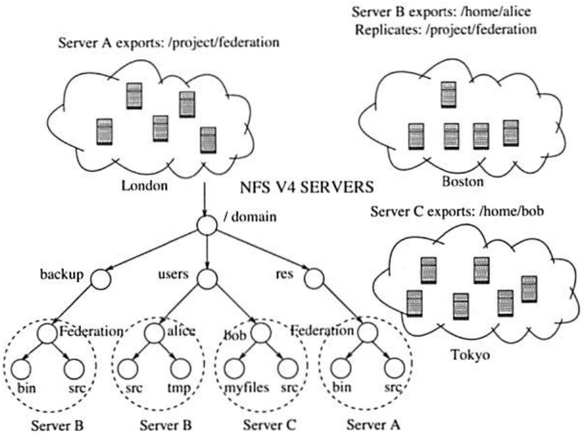
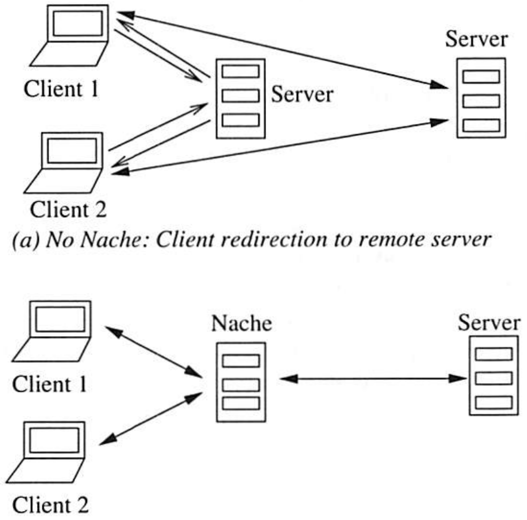


Figure 4: *Wide-area Federation of Filesystems*



(a) No Nache: Client redirection to remote server

(b) With Nache: Client access directly from proxy

Figure 5: *System Architecture (a) Client redirection without NFSv4 Proxy (b) With Nache*

The client redirection model relies on the standard NFSv4 client's ability to follow the protocol-specified referral. A client first mounts the root of the namespace from a local NFSv4 server. When it traverses a directory (filesystem) that happens to refer to a remote server location, the server initiates redirection and, when queried, returns an ordered list of remote server addresses and path names. The client then "sub-mounts" the directory (filesystem) from one of the remote servers in the list and continues its traversal.

Two factors contribute to the overhead of handling a redirection: (i) the processing overhead of following the referral along with the new sub-mount at the client, and (ii) the network overhead of accessing data from a remote server, possibly over a lower bandwidth, higher latency link. Figure 6 captures the processing overhead of following the referral. It shows that the time taken for

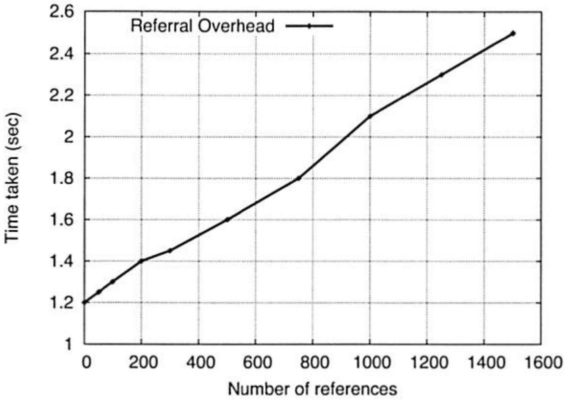


Figure 6: *Redirection overhead in traversing (ls -lR) a directory tree with 1500 directories. X-axis shows the number of redirections; Y-axis shows the response time of a tree traversal. All redirections point to the same path on the remote server.*

traversing an NFS mounted directory tree (with ls -lR) with 1500 directories and no re-directions is around 1.2 seconds. The time for a similar traversal where each directory is a referral to another server is 2.5 sec. In this example, all the referrals point to the same path on another server, thereby requiring only a single submount. As more servers get added to the federation, the time would further increase due to the additional submounts.

The network overhead of a redirection is, as expected, due to the latency and delay caused by a remote data transfer. For example, we measured the time taken to read a file of 8MB when the redirection was to a local (same LAN) server to be 0.7 secs while that when the redirection was to a remote server (over the WAN with the client in CA and the server in NY) was 63.6 secs.

When a number of co-located clients mount a filesystem from a local server they may each incur the redirection and remote access overhead. To reduce network latency, one obvious approach is to replicate the data at multiple servers and let the client select the closest replica location. However, replication is not feasible for all workloads as data may be constantly changing and what data needs to be replicated may not always be known.

Ideally, the data should be available locally on demand, kept consistent with respect to the remote server, and shared among all the local clients. The local server can act as a proxy by caching the remote data and forwarding requests that cannot be serviced locally to the remote server.

4.2 Nache Architecture

Conceptually, the Nache proxy is similar to any other proxy say a web caching proxy. However, a number of factors make it more challenging. First, NFSv4 is a stateful protocol with the server maintaining open state, clientids, lock owners, etc. Second, unlike web caches, data is both read and written and the close-to-open consistency guarantee has to be maintained. Third, other considerations of file handle management and security concerns make a file server proxy non-trivial.

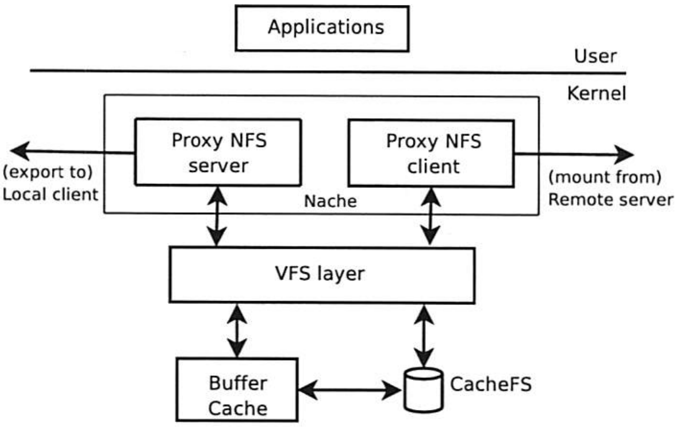


Figure 7: *Nache Architecture Block Diagram*

Observe that the Nache proxy is different from a network layer-7 switch based routing proxies [4]. Such routing proxies typically terminate the TCP connection, parse the NFS packet information and route the packet to the appropriate server. They do not act as a fully functioning NFS server and client. The caching done in routing proxies is for read-only data where consistency support is not a concern. Moreover, the routing switch/appliance becomes a single bottleneck for all data to and from the client.

The Nache caching proxy relies on the delegation support in NFSv4 for improving cache performance. The delegation granted by the remote server to the proxy is shared among the local clients of the proxy. Thus, all operations on a file across multiple clients can be handled locally at the proxy. As we discussed in Section 3, in scenarios where a number of local clients are sharing a file for reads and writes, the overhead of a delegation recall is prohibitive. With the shared delegation model of Nache, a recall is avoided if all accesses are from local clients. If there is absolutely no sharing, however, it is better for clients to directly receive the delegations.

Figure 7 shows the block components of the Nache proxy architecture. It consists of the NFS server and client components that communicate via the VFS layer. CacheFS [19] is used to add persistence to the cache. Nache fits well in the NFSv4 model of a federated filesystem where each server exports the root of the common namespace and can act as a proxy for a remote server. On entering a directory that is not local, a server can either redirect the client to a local replica if data is replicated or act as a proxy to fetch the data locally.

5 Implementation

In essence, Nache acts as a bridge between NFSv4 clients and servers, handling client requests and forwarding them to target NFSv4 servers when there is a cache miss. It provides all the server functionality that NFS clients expect. Nache has been implemented on Linux by gluing together the client and server code paths for the various NFS operations. The implementation centers around three main areas: *Cascaded NFS mounts*, *NFS operation forwarding* and *Sub-operation RPC call handling*.

5.1 Cascaded Mounts

Nache enables an NFS client to access a remote NFSv4 filesystem locally by mounting it from the proxy. Internally, Nache mounts the remote filesystem and re-exports it so that local clients can access it over the NFS protocol. To provide such cascaded mounts, Nache must be able to export NFSv4 mounted filesystems. For this, we define *export operations* for the NFS client that allow client requests to be routed to remote servers.

We use Linux *bind* mounts to link the filesystem mounted from the remote server with a directory within the root of the exported pseudo filesystem at the proxy. Consider the case where a remote NFSv4 server `nfs4-server` is exporting a filesystem at `/export`, while `/nfs4` is the root of the pseudo filesystem exported by the proxy `nfs4-proxy`. In order for `nfs4-proxy` to re-export `/export`, we need to bind mount `/export` to a directory in the tree rooted at `/nfs4`, say at `/nfs4/export`. Here `/nfs4` is the root of the proxy's NFSv4 pseudo filesystem. This can be done at the proxy by the following sequence of commands:

```
    mount -t nfs4 nfs4-server:/ /export
    mount --bind /export /nfs4/export
```

The client can then access the exported filesystem from the proxy as:

```
    mount -t nfs4 nfs4-proxy:/ /nfs
```

With cascaded mounts, the client can mount the remote server's filesystem from the proxy and access it locally at `/nfs/export`. We ensure that the proxy exports the remote filesystem using the appropriate options (`nohide, crossmnt`) that enable the NFSv4 client to view a filesystem mounted on another filesystem. Thus the modified code is able to export a remote filesystem that is mounted over NFS. The proxy is implemented by merging the functionality of the NFSv4 client and server kernel modules which communicate through an unmodified VFS layer. The interaction of the client and server components is shown in Figure 8. The NFSv4 client sends an RPC request to the proxy's server-side module (nfsd in Linux). The server-side module at the proxy forwards the call to the proxy's client-side module (nfs in Linux) using the VFS interface. Finally, the client-side module at the proxy forwards the call to the remote server if needed. The response from the remote server is stored at the client-side buffer cache and can be reused for later requests.

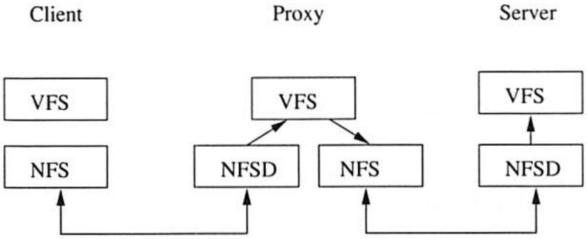


Figure 8: *Communication between kernel modules at client, proxy and server.*

The official Linux NFS kernel implementation does not permit re-exporting of NFS mounted filesystems (multi-hop NFS) because it is difficult to detect errors such as infinite mount loops. Moreover, there are concerns over host or network failures, access control and authentication issues along with the inefficiency of using an intermediate server between the NFS client and the file server in a LAN environment [36]. In a trusted enterprise environment with WAN access, however, multihop NFS can be used to reduce the high latency by caching locally at the proxy. NFSv4 also has better security, authentication and failure handling support that can be leveraged by the proxy. We discuss the security issues later in Section 5.5.

5.2 NFS Operation Forwarding

Once an NFS operation (such as LOOKUP or OPEN) is received at the Nache server-side module (Nache server), it is forwarded to the client-side module (Nache client) via the VFS interface. Thus the NFS request gets translated to a VFS call which then calls the corresponding operation in the underlying local filesystem. In the proxy case, this is the *NFS mounted* remote filesystem.

The translation between an NFS request to a VFS call and back to an NFS request works without much modification in most cases. However, operations that are "stateful" require special handling both at the Nache server and at the Nache client. Additionally, calls from the Nache server to the VFS layer need modifications to make them appear to have originated from a local process at the proxy. In the following discussion we will describe in detail some of the operations that need special attention.

- *OPEN:* The OPEN request processing required modifications due to the integrated handling of NFSv4 OPEN and LOOKUP operations in Linux. In the NFSv4 client, processing a file OPEN triggers a LOOKUP operation and the OPEN request is sent to the server as part of LOOKUP. The NFS client's generic file open function (nfs_open) does not actually send an OPEN request to the server. This created a problem at the proxy during an OPEN request because the Nache server was invoking the generic filesystem open function through the VFS layer which translated to nfs_open for an NFS mounted filesystem. This function, however, would not send an OPEN request to the remote server.

To resolve this, we modified the open operation in the Nache server to emulate the way a local file open would have been seen by the Nache client. In Linux, this includes extracting the arguments of the open and calling the appropriate lookup function in the Nache client. The NFSv4 server stores state (nfs4_stateid) associated with a file OPEN request. To obtain this state at Nache, we modified the return path of the OPEN to extract the relevant state created at the Nache client and populate the stateid structure in the Nache server.

- *CREATE:* In the normal Linux NFS client, the create function (nfs_create) needs some data (nameidata) that is populated by the VFS layer. In Nache, when translating between the server-side and client-side functions, the information to do this initialization is not readily available. In the Nache server, on a create request, we extract the open flags and the create mode and use it to initialize nameidata before invoking the VFS function, vfs_create(). That, in turn, calls the Nache client's nfs_create() with the required data already initialized.
- *LOCK:* When a LOCK request is processed at the Nache server, it gets translated to the underlying POSIX function. The POSIX lock function, however, does not call the associated NFS locking (nfs_lock) function in the Nache client. This is because most regular filesystems do not specify their own lock operation. Hence, none of the lock requests (LOCK, LOCKU) would get forwarded to the remote server. We modified the lock function in the Nache server to check if the inode on which the lock is desired belongs to an NFS mounted filesystem and call the corresponding lock operation of the Nache client.
- *CLOSE:* This happens to be one of the most complicated operations to handle. The CLOSE operation depends on the state associated with the file at the client. A client with multiple OPENs for the same file only sends one CLOSE to the server (the last one). At the Nache server, there are two scenarios that could occur: (i) the multiple opens are all from the same client, (ii) the multiple opens are from different clients. Although the number of CLOSE requests received by the Nache server is equal to the number of distinct clients, the Nache client should only send one CLOSE to the remote server. To handle a CLOSE, we have to keep track of the state associated with a file and make sure that the file open counters are adjusted properly during the OPEN and CLOSE operations. The counters should be incremented only once per client and not for every OPEN request from the same client. This client state is difficult to maintain at the Nache server. In our current implementation, we experimented with various options of counter manipulation and eventually chose to simply count the number of open requests seen by the proxy across all clients. This implies that some CLOSE requests may not be sent to the remote server in cases where a client opens the same file multiple times. We expect to provide a better solution as we continue the development of Nache.

5.3 Sub-operation RPC Calls

Another issue that arose due to performance-related optimizations is that the NFS server while handling a request does low level inode operations which, in Nache, result in RPC calls sent to the remote server from the proxy. The NFS server is optimized for doing local filesystem operations and not for remote filesystem access. As a representative example, consider the OPEN request at the server. In processing the OPEN (nfsd4_open), the server calls other local operations for lookup (do_open_lookup) and permissions checking (do_open_permissions). In Nache, however these operations cannot be done locally and are sent to the remote server. Thus a single OPEN call leads to three RPC calls exchanged between Nache and the remote server. To avoid this, we reorder these checks in the usual path in the Nache server code as they would be eventually done in the OPEN processing at the remote server. Numerous such optimizations are possible to reduce the number of remote accesses.

5.4 CacheFS and Persistence

To add persistence to the cache, increase cache capacity, and survive reboots, Nache relies on CacheFS [19]. CacheFS is a caching filesystem layer that can be used to enhance the performance of a distributed filesystem such as NFS or a slow device such as a CD-ROM. It is designed as a layered filesystem which means that it can cache a *back* filesystem (such as NFS) on the *front* filesystem (the local filesystem). CacheFS is not a standalone filesystem; instead it is meant to work with the front and back filesystems. CacheFS was originally used for AFS caching but is now available for any filesystem and is supported on many platforms including Linux. With CacheFS, the system administrator can set aside a partition on a block device for file caching which is then mounted locally with an interface for any filesystem to use. When mounting a remote NFS filesystem, the admin specifies the local mount point of the CacheFS device. In Nache, CacheFS is used primarily to act as an on-disk extension of the buffer cache.

CacheFS does not maintain the directory structure of the source filesystem. Instead, it stores cache data in the form of a database for easy searching. The administrator can manually force files out of the cache by simply deleting them from the mounted filesystem. Caching granularity can range from whole files to file pages. CacheFS does not guarantee that files will be available in the cache and can implement its own cache replacement policies. The filesystem using CacheFS should be able to continue operation even when the CacheFS device is not available.

5.5 Discussion

Security Issues In a proxy setting, security becomes an important concern. The proxy can provide clients access to read and modify data without the server knowing or verifying their authority to do so. The current implementation of Nache does not include any additional security support beyond what exists with vanilla delegations. Presently, the server returns the list of access control entries when a delegation is awarded. However, the user ID space at the proxy and server maybe different. In such cases the proxy resorts to sending an ACCESS request to the server to verify the access permissions for that client for every OPEN. In case Nache is deployed as part of a managed federation, the client access control can be centrally managed.

Delegation Policy Another issue that needs attention is the policy used by the server to decide when to award a delegation. For example, the current implementation in Linux awards a delegation on the second OPEN by the same client. This policy may be too liberal in giving out delegations which must be recalled if there is a conflicting access. It may not be feasible to implement a complex policy based on access history of each file within the kernel. As part of the Nache implementation, we are exploring different policies that try to maintain additional access information about the file to better grant delegations.

Protocol Translation One interesting use of Nache is as a protocol translator between NFS versions. Nache behaves exactly like a v4 server to v4 clients and as a v3 server to v3 clients. This is useful when leveraging client features in NFSv4 such as redirection and volatile file handles and the WAN-friendly protocol features. The file delegation happens between the proxy acting as a v4 client and the back-end v4 server, hence the consistency semantics are maintained for both v3 and v4 clients. Efforts to integrate with an NFSv3 client are ongoing in Nache.

Failure Handling Any component of the system — client, proxy, or server can fail during an NFS operation. In general the failure of the server can be masked somewhat by the proxy as it can respond to reads from the cache (although these can no longer be backed by delegations). However, the failure of the proxy will affect the clients in two ways. First, the clients will not be able to access the data on the server even when the server is operational. Second, the writes done at the proxy that have not been flushed back to the server may be lost. Some of the problems of write delegations to a client without a proxy are further complicated with the use of a proxy as dirty data could have been read by multiple clients. If the writes are flushed periodically to the server the lag between the proxy and server state can be reduced.

6 Experimental Evaluation

The Nache implementation is based on CITI, University of Michigan's NFSv4 patches [13] applied to Linux kernel version 2.6.17. User-space tools required on Linux (such as nfs-utils) also have CITI patches applied for better NFSv4 capabilities. We used IBM xSeries 335 servers with Intel Pentium III (1133MHz) processor, 1GB of RAM, 40GB IDE disk with the ext3 filesystem for our experimental testbed. For the WAN access experiments, we used machines over a wide area network between IBM Almaden (California) and IBM Watson (New York) that had a round-trip ping delay of about 75 msec. One of the local machines was set up as the Nache proxy that runs the kernel with *nfs* and *nfsd* kernel modules suitably modified for proxy implementation. The remote server machine's *nfsd* module is also modified to enable write delegations and provide a fix for the COMMIT operation on a delegated file, as we discuss later in this section.

The evaluation is divided into four categories. First, we evaluate the delegation support currently in Linux. Next, we experiment with write delegations and their performance with certain workloads. We argue that write delegations should be awarded more carefully. Third, we test Nache with a set of workloads for gains achieved in terms of the total NFS operations sent to the server and the time taken to complete certain file operations. Some of the workloads are based on the different profiles available in Sun's filesystem benchmark Filebench [24]. The setup consists of two or more clients, a Nache proxy and one server. We compute the benefits with Nache in scenarios where clients show overlap in their data access. Finally, we measure the overhead of deploying Nache especially in scenarios where there is no sharing among clients, thereby, limiting the benefits of a shared cache.

6.1 Effect of Delegations on NFSv4 Operations

We discussed the advantages and shortcomings of the NFSv4 delegation support in the Linux kernel in Section 3. Here we further investigate how individual file operations are affected when a delegation is granted. Recall that a delegation is awarded to a client when a callback channel has been successfully established between the server and the client for a recall; and a client opens a file twice (not necessarily the same file). As mentioned in Section 3, presently the Linux server only hands out read delegations, while we added a prototype implementation for awarding write delegations. Our prototype uses the same policy as used by read delegations, namely a write delegation is granted on the second OPEN for write from the same client. With our changes, we observed that the server did not recall delegations correctly on a conflicting OPEN and was not handling a COMMIT from a client with write delegations properly. The server tried to recall the delegation on a COMMIT which the client would not return until the COMMIT succeeded. This deadlock caused new OPENs to see long delays. Our prototype fixes the delegation recall handling on a conflicting OPEN and the COMMIT on a delegated file. We expect these issues to be correctly resolved when write delegations are officially supported in the Linux kernel.

NFS Op.	Delegation ON	Delegation OFF
OPEN	101	800
CLOSE	101	800
ACCESS	101	1
GETATTR	1	700
LOCK	101	800
LOCKU	1	800
READ	100	100

Table 2: *The number of NFSv4 operations seen by the server with and without read delegations.*

To assess activity locally and at the server in the presence of delegations, we performed a sequence of OPEN-READ-CLOSE operations over 100 files and repeated it 8 times. Table 2 shows the operations with read delegations enabled. Observe that there are 101 OPEN (and CLOSE) operations that are sent to the server with read delegations. This is because the first OPEN from the client does not get a delegation as per the server decision policy and is sent again to the server on the second run. All other files only require one OPEN to be sent to the server. However, as per the protocol, the client must check caller's access rights to the file even when delegations are available (after the attribute timeout has expired). We detect one such timeout during our run, hence observe 101 ACCESS calls in the presence of delegations. Note also that no GETATTR requests need to be sent to the server for revalidation when read delegations have been obtained. Also the number of reads sent to the server are the same with and without delegations as the file data is cached after the first OPEN in both cases. Similarly, we introduced the LOCK-ULOCK pair in the sequence to determine the LOCK operation behavior and observe that unlocks can be serviced locally when delegations are available.

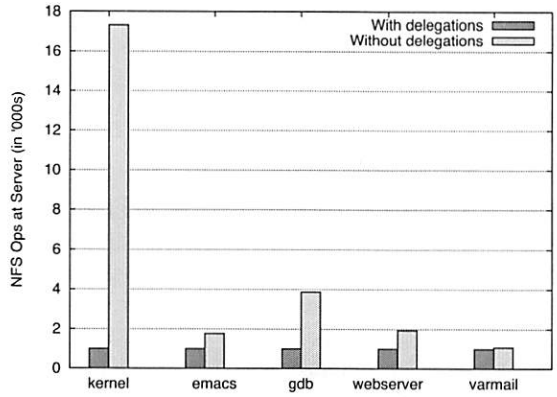


Figure 9: *Performance of Read and Write delegations (total ops at server): the Y-axis shows the server ops with and without delegations.*

We repeated the experiments with write delegations and found that all the writes are flushed to the server on a CLOSE although they can be kept dirty if delegations are in place. The inconsistent results are due in part to the fact that the Linux server's delegation implementation is incomplete.

To further study the benefits of delegations, we used different workloads to evaluate the performance. One set of workloads consist of compilations of different source code packages, namely the Linux kernel, Emacs and GDB, where the source tree is mounted over NFS from a remote server. Another set includes webserver and varmail profiles from the *filebench* benchmark. Figure 9 shows the server operations for different workloads with read and write delegations enabled. Here the number of operations at the server are 16 to 1.2 times lower for the different workloads when delegations are granted. Notice the massive benefits in the compile of the Linux kernel compared to those of Emacs and GDB. This is because the kernel is relatively self-contained and does not reference many files outside the source tree. On the other hand, the compiles of Emacs and GDB use standard include files and libraries during the build which are accessed from the local filesystem. Figure 10 shows the benefits in terms client response time and throughput for the same workloads.

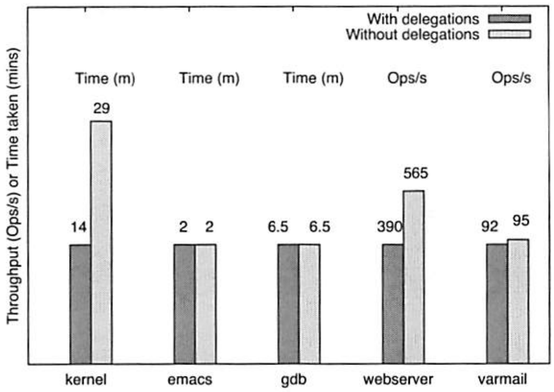


Figure 10: *Performance of Read and Write delegation (total throughput): the Y-axis shows the server throughput (ops/sec) or response time with and without delegations. The number at the top of the bar is the actual value.*

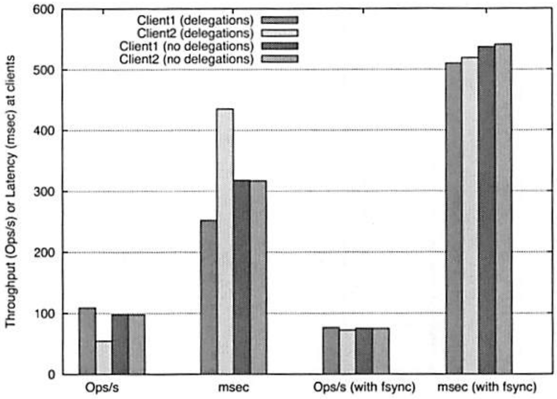


Figure 11: *Effect of WRITE delegation on a workload with write sharing among clients. The 4 bars in each set show the performance of Client1 with write delegation, Client2 with a conflicting write and Client1 and Client2 with delegations disabled respectively.*

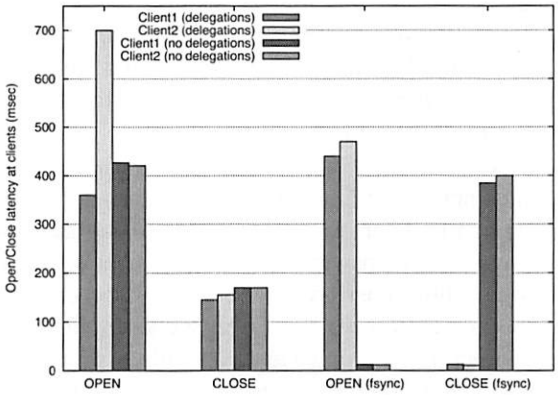


Figure 12: *Effect of WRITE delegation on latencies of OPEN and CLOSE operations.*

6.2 Impact of Write Delegation Policy

In Section 3, we showed that the overhead of a delegation recall can be substantial enough to negate its benefits. Here, we further investigate the impact for delegation recalls in workloads with concurrent writes. To simulate such a workload, we modified Filebench to use multiple threads each doing a sequence of open-read-append-close operations on a set of files directly from the server. Because of write delegations being awarded and recalled, the performance varies by a large degree depending on the number of clients. With two clients doing concurrent writes, one client's throughput was observed to be 109 ops/sec while the other client's was half that at 54 ops/sec. Similarly, the average latencies observed by the clients were 252 ms and 435 ms, respectively. With write delegations disabled, both the clients had similar performance. The first two bars show these results in Figure 11.

We had also shown in Section 3 that the time taken for a conflicting OPEN increases substantially with the amount of dirty data that needs to be flushed on a delegation recall. However, if the dirty data is synced at regular intervals, the time taken to respond to a conflicting OPEN should be significantly reduced. To verify this we added a `fsync()` operation before the `close()` in the Filebench workload for each file.

This improved the throughput and latency for the second client and both achieved 75 ops/sec with average latencies of 531ms and 548ms respectively. These results are shown in the last two bars of Figure 11. Keeping the dirty data size less than 1MB seems to be a good heuristic as the overhead remains fairly constant when the unwritten data ranges from a few KBs to 1 MB. Note that latencies are worse with `fsync()` because data is written more frequently.

Similarly, Figure 12 shows the average latency for OPEN and CLOSE operations for each of the two clients in the presence of write delegations and with the optional periodic `fsync()`. We observe that the second client's OPEN takes twice the amount of time as the first client when write delegations are handed out. This reenforces the claim that the delegation policy can have an adverse affect on the performance if there are conflicting accesses.

6.3 Performance Benefits of Nache

In this section we evaluate the performance benefits of Nache using different workloads. In the experimental testbed, a group of client machines access a remote file server via the same proxy server. For results demonstrating LAN access, all machines are on the same 100 Mbps LAN. For experiments with WAN access, the access is over an enterprise network between California and New York.

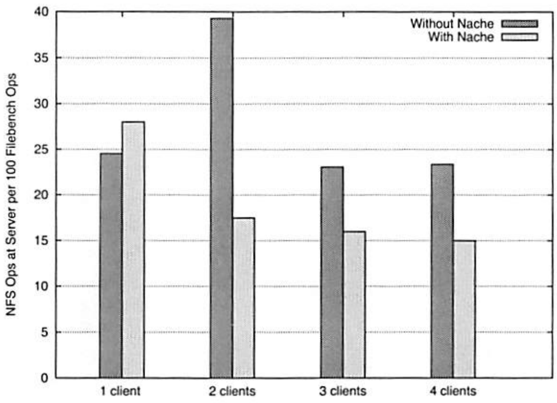


Figure 13: *Benefits of Nache: Filebench Web server workload.*

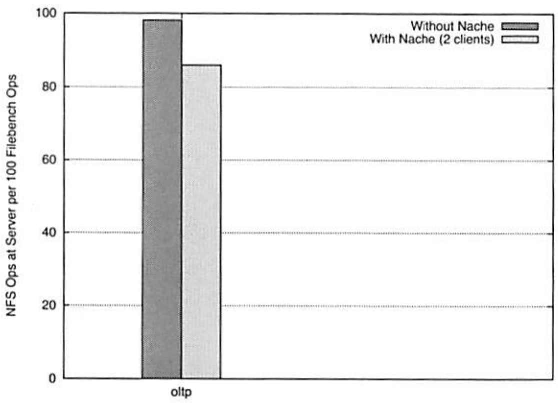


Figure 14: *Benefits of Nache: Filebench OLTP workload.*

6.3.1 Filebench Workloads

The Filebench benchmark contains a number of profiles that are representative of different types of workloads. Some of them are: a webserver with read-only data, an OLTP transaction processing system, a *varmail* workload (similar to the Postmark benchmark), and a web proxy workload (*webserver*). We provide results for the webserver and OLTP workloads.

Webserver: This workload generates a dataset with a specified number of directories and files using a gamma distribution to determine the number of sub-directories and files. It then spawns a specified number of threads where each thread performs a sequence of open, read entire file and close operations over a chosen number of files, outputting resulting data to a logfile. As part of the runs, we observed that Filebench was accessing all files uniformly with no skew which we believe is not representative of a typical webserver access pattern [11]. We modified the access pattern function to select files based on a Zipf distribution [11]. We obtained results for both the uniform and Zipf file access pattern using 500 files, 10 threads and a run time of 100 seconds. Figure 13 shows the total number of operations sent to the server normalized with respect to the total number of Filebench operations. The normalization was done to remove the effect of any variance in the total number of operations generated by filebench as the number of clients varied. We observe that Nache reduces the number of operations seen at the server by 38% with four clients.

OLTP: The OLTP workload is a database emulator using an I/O model from Oracle 9i. This workload tests for the performance of small random reads and writes. In our experiments we use 20 reader processes, 5 processes for asynchronous writing, and a log writer. Since Filebench was originally written for Solaris and modified to work on Linux, we found that it was quite unstable in running the OLTP profile, possibly due to the asynchronous I/O requests. We could not reliably run the OLTP workload for more than two clients. Figure 14 shows the total number of operations sent to the server normalized with respect to the total number of filebench operations. Observe that with Nache, the server operations are reduces by 12.5% for two clients.

6.3.2 Software Builds

In this experiment we consider the scenario of a joint software development project where clients build individual versions of a large software package for local testing. This involves using a common source tree at a remote server and building it with locally modified files. We performed the build of three software packages: Linux kernel (version 2.6.17), GDB (version 6.5), and Emacs (version 21.3). In each case, the directory containing the source was NFS mounted at the proxy and the object and executable files generated during the compilation are written locally at the client.

Figures 15, 16, 17 show the number of NFS operations sent to the server for each of the three workloads with varying number of clients sharing the proxy cache. Based on these experiments we observe that: (i) with Nache, the operations at the server decrease or stay flat as the number of clients increase (i.e., there is more sharing), (ii) without Nache, the operations at the server linearly increase with the number of clients (as do the operations at the proxy with Nache), (iii) with Nache, the time taken over a LAN for various builds stays constant as the number of clients increase (iv) with Nache, the time taken for various builds decreases over WAN (as shown in Figure 18). For example, for the kernel build in Figure 15, the server operations are reduced by more than 50% with Nache and 2 clients. Interestingly, with Nache the number of operations at the server with multiple clients is sometimes less than that for a single client. This is because some operations such as CLOSE will not be sent to the server if multiple clients have opened the file via Nache. Furthermore, the number of granted

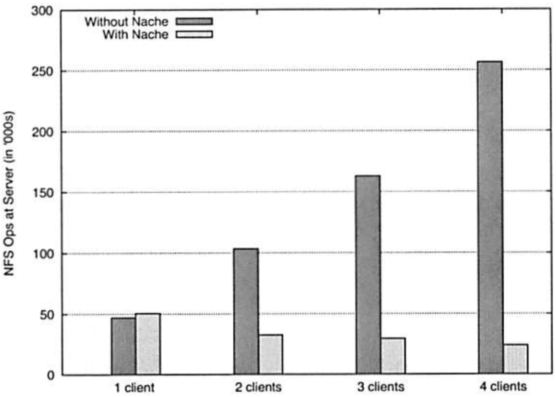


Figure 15: *Benefits of Nache: Compile of Linux kernel.*

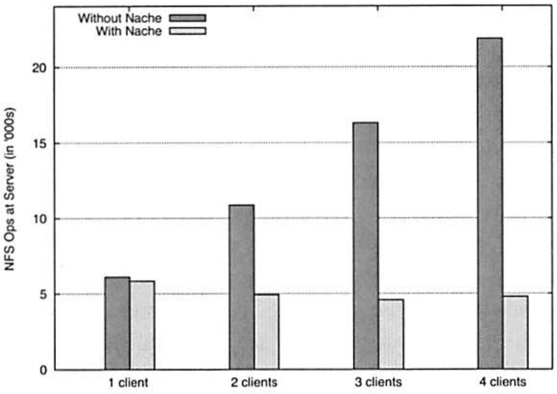


Figure 16: *Benefits of Nache: Compile of GDB.*

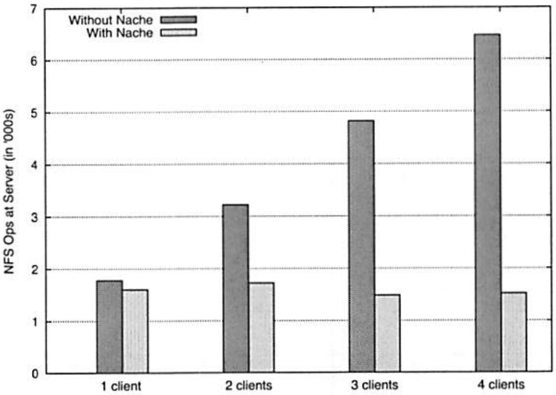


Figure 17: *Benefits of Nache: Compile of Emacs.*

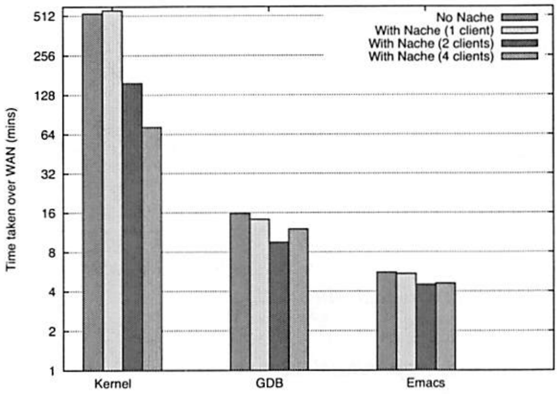


Figure 18: *Effect of Proxy on response time over a WAN.*

delegations is higher as more files may be opened concurrently with two clients than with one client. This is an artifact of the way the server awards delegations (on the second concurrent OPEN from a client) rather than an inherent benefit of the proxy. Similarly, the response time on a WAN reduces by 70% for the same build. Observe that the kernel compile had the best performance improvement when using a proxy for the same reasons as discussed in Section 6.1.

Response time over WAN We repeated the software build experiments over a WAN to measure the latency improvements with Nache. Figure 18 shows the time taken to build the Linux kernel, GDB and Emacs with and without Nache. As in the LAN case, the source code is NFS mounted, but the object files and executables are stored locally.

For a single client accessing data directly from the remote server without Nache, the time taken is 533min, 15.8min and 5.6min respectively. The response time decreases as we increase the number of clients going through Nache. In case of the kernel build, the response time is slightly higher with a single client due to the overhead incurred by an intermediary proxy and the absence of sharing. With two clients, on the other hand, the response time is 3.5 times lower than without Nache. The marked improvement in response time is due in part to the fewer operations sent to the server as we discussed earlier.

6.4 Measuring Proxy Overhead

In certain scenarios such as a single client accessing files or when there is no file sharing, Nache simply adds to the data path without any added benefit of a shared consistent cache. In this section, we measure the overhead of the proxy in such scenarios using micro-benchmark tests. The workloads used for the micro-benchmark are as follows:

- *Create Files:* Creates 25,000 files, writes up to 16KB of data and then closes all of them.
- *Random Reads:* Performs random reads on a large 1GB file using a single thread.
- *Random Writes:* Performs random writes on a large 1GB file using a single thread.

Micro-benchmark	ops/sec (Nache)	ops/sec (NFSv4)	latency (Nache) ms	latency (NFSv4) ms
Create Files	37	40	1283	1172
Random Reads	99	127	1.3	7.8
Random Writes	27	32	37.6	31.3
Random Appends	9	11.9	220	146
Create Files (WAN)	9	16	5217	3009
Random Reads (WAN)	77	48	12.9	20.7
Random Writes (WAN)	8.6	10	116.3	98
Random Appends (WAN)	2.2	2.4	883.4	821

Table 3: *Micro-benchmark results and comparison among configurations with and without Nache*

- *Random Appends:* Creates a dataset with multiple files and does append-fsync on the files with random append sizes ranging from 1KB to 8KB. The file is flushed (fsync) every 10MB up to the maximum file size of 1GB.

We ran all the benchmarks over a LAN and a WAN to measure latencies for the different workloads. Table 3 shows the operation rate (ops/sec), the number of NFS operations sent to the server and the average latency for each of the micro-benchmarks. The results show that overhead is higher (7-40% worse) in create-intensive cases compared to the other scenarios. This is expected because Nache just acts as another router and seems to provide no benefit in terms of data or metadata caching. Some gains are observed in case of random reads on WAN which can be attributed to caching at the proxy. The Nache code is not fine-tuned and some of the overhead should decrease with some optimizations.

7 Related Work

Caching has always been used in distributed filesystems to improve performance. Most popular distributed filesystems rely on a client-server architecture where caching is done primarily at the client. While the various NFS versions have supported client-side caching, they enforce only weak cache consistency. NFS extensions such as Spritely-NFS and NQNFS tried to improve the NFS consistency semantics. Spritely-NFS [35] used the Sprite [27] cache consistency protocols and applied them to NFS. This allowed for better cache consistency by using server callbacks. NQNFS [23] also aimed at improving NFS consistency semantics but differed from Sprite in the way it detected write sharing.

While NFS was more suited for LAN access, the AFS [2, 17, 18] filesystem was designed for wide-area access. For this, AFS relied extensively on client-side file caching and supported cache consistency through callbacks. The successor to AFS was the DFS [20] filesystem which had most of the features of AFS but also integrated with the OSF DCE platform. DFS provided better load balancing and synchronization features along with transparency across domains within an enterprise for easy administration. AFS also led to the Coda [22] filesystem that dealt with replication and client-side persistent caching for better scalability while focusing on disconnected operations.

Along with NFS and AFS, which are more prevalent on Unix platforms, Microsoft Windows clients use the CIFS (Common Internet File System) [34] protocol to share data over a network. CIFS provides various optimizations such as batched messages, opportunistic locks for stronger cache coherency, and local buffering to improve response times and save network round trips. The Microsoft DFS filesystem leverages the CIFS protocol to create a filesystem federation across multiple hosts [25].

In case of AFS, along with client-side caching, Muntz-Honeyman [26] analyzed the performance of a multi-level cache for improving client response times in a distributed filesystem. They concluded that multi-level caching may not be very useful due to insufficient sharing among client workloads. While it is known that the effectiveness of an intermediate cache is limited by the degree the sharing across clients, we believe that remote collaboration has significantly increased in the last decade due to advances in network bandwidth and improvements in collaborative tools. Current web workloads, for example, show a high degree of sharing of "hot" documents across clients [11]. Similarly, distributed collaborative projects have increased with global outsourcing. In Nache, we show that even when sharing is low (say 8-10%), the gain in response time can be high when data is accessed across a WAN. Moreover, Muntz-Honeyman's paper shows that an intermediate proxy can substantially reduce the peak load at the server. Thus, along with client response time, a proxy can also improve the server scalability by reducing server overload. We observed with Nache that even a very low degree of sharing can eliminate all the gains of a pure client-side cache due to the recall of delegations on a conflicting access. This suggests that a shared proxy cache is beneficial to reduce conflicts if cache consistency is desired.

The client-side cache can also be optimized by making it persistent and policy-based. Nache relies on CacheFS for on-disk caching. Xcachefs is similar to CacheFS in that it allows persistent caching of remote filesystems but further improves performance by de-coupling the cache policy from the underlying filesystem [33]. It allows clients to augment the caching mechanism of the underlying filesystem by specifying workload specific caching policies.

Recently a plethora of commercial WAFS and WAN acceleration products have started offering caches for NFS and CIFS protocol for improving wide-area performance. These often use custom devices both in front of the server and the client with an optimized protocol in between [5, 3].

Although proxy caching is not that prevalent in file serving environments, it has been widely used in the Web due in part to the read-only nature of the data and the high degree of WAN accesses. The Squid proxy cache that grew out of the Harvest project [14, 15] uses a hierarchical cache organization to cache FTP, HTTP and DNS data.

While Nache focuses on improving the wide-area access performance of existing file servers, numerous research efforts have focused on building scalable file servers. Slice [7] implements a scalable network-wide storage by interposing a request switching filter on the network path between clients and servers. Clients see a unified file volume and access it over NFS. Slice is mainly designed to provide a scalable, powerful NAS abstraction over LAN whereas our main goal is to improve file serving performance over a WAN. The Tiger file server [10] provides constant rate delivery by striping the data across distributed machines (connected via high speed ATM) and balancing limited I/O, network and disk resources across different workloads. Farsite [6] implements a server-less distributed filesystem that provides the benefits of shared namespace, location transparency and low cost. Thus it transforms unreliable local storage at clients to a more reliable, logically centralized storage service. xFS is another server-less distributed filesystem that uses cooperative caching to improve performance [8]. Lustre [1] is an object based distributed filesystem that is designed to work with object based storage devices where controllers can manipulate file objects. This leads to better I/O performance, scalability, and storage management. While these and other efforts [31, 30] have focused on improving file serving performance, they are not designed for improving the performance of existing file servers and NAS appliances.

8 Conclusion

In this paper, we have presented the design and implementation of a caching proxy for NFSv4. Nache leverages the features of NFSv4 to improve the performance of file accesses in a wide-area distributed environment. Basically, the Nache proxy sits in between a local NFS client and a remote NFS server caching the remote data closer to the client. Nache acts as an NFS server to the local client and as an NFS client to the remote server. To provide cache consistency Nache exploits the read and write delegations support in NFSv4. We highlighted the three main contributions of the paper. First, we explored the performance implications of read and write open delegations in NFSv4. Second, we detailed the implementation of the Nache proxy cache architecture on the Linux 2.6 platform. Finally, we discussed how to leverage delegations to provide consistent caching in the Nache proxy. Using our testbed infrastructure, we demonstrated the performance benefits of Nache using the *Filebench* benchmark and different workloads. In most cases the Nache proxy can reduce the number of operations seen at the server by 10 to 50%.

As part of on going work we are exploring different policies for awarding read and write delegations to lower the probability of a conflict. Also the Nache architecture is being integrated with the federated filesystem architecture that provides a common file-based view of all data in an enterprise.

References

[1] Cluster File Systems Inc., Lustre: A Scalable, High-Performance File System. http://www.lustre.org/docs/whitepaper.pdf.

[2] OpenAFS: http://www.openafs.org.

[3] Packeteer Inc., Wide Area File Services: Delivering on the Promse of Storage and Server Consolidation at the Branch Office. http://www.packeteer.com/resources/prod-sol/WAFS_WP.pdf.

[4] Acopia Networks, Intelligent File Virtualization with Acopia. http://www.acopianetworks.com/pdfs/adaptive_resource_networking/Intelli%gent_File_Virtualization_wp.pdf, Nov. 2006.

[5] Expand Networks, WAN Application Acceleration for LAN-like Performance. http://www.expand.com/products/WhitePapers/wanForLan.pdf, Nov. 2006.

[6] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment, Dec 2002.

[7] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed Request Routing for Scalable Network Storage. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, Oct 2000.

[8] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *"Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles"*, Dec 1995.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] W. Bolosky, J. Draves, R. Fitzgerald, G. Gibson, M. Jones, S. Levi, N. Myhrvold, and R. Rashid. The Tiger Video Fileserver. In *Proceedings of the 6th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'96)*, Apr 1996.

[11] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. IEEE INFOCOM*, pages 126–134, 1999.

[12] B. Callaghan. *NFS Illustrated*. Addison-Wesley Longman Ltd., Essex, UK, 2000.

[13] CITI. Projects: NFS Version 4 Open Source Reference Implementation. `http://www.citi.umich.edu/projects/nfsv4/linux`, Jun 2006.

[14] J. Dilley, M. Arlitt, and S. Perret. Enhancement and validation of the Squid cache replacement policy. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

[15] D. Hardy and M. Schwartz. Harvest user's manual, 1995.

[16] J. Haswell, M. Naik, S. Parkes, and R. Tewari. Glamour: A Wide-Area Filesystem Middleware Using NFSv4. Technical Report RJ10368, IBM, 2005.

[17] J. Howard and et al. An Overview of the Andrew Filesystem. In *Usenix Winter Techinal Conference*, Feb 1988.

[18] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[19] D. Howells. FS-Cache: A Network Filesystem Caching Facility. In *Proceedings of the Linux Symposium*, volume 1, Jul 2006.

[20] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Buttos, S. Chutani, C. F. Everhart, W. A. Mason, S. Tu, and E. R. Zayas. DEcorum File System Architectural Overview. In *Proceedings of the Summer USENIX Technical Conference*, 1990.

[21] S. Khan. NFSv4.1: Directory Delegations and Notifications, Internet draft. `http://tools.ietf.org/html/draft-ietf-nfsv4-directory-delegation-01`, Mar 2005.

[22] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[23] R. Macklem. Not Quite NFS, Soft Cache Consistency for NFS. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 261–278, San Fransisco, CA, USA, 1994.

[24] R. McDougall, J. Crase, and S. Debnath. FileBench: File System Microbenchmarks. `http://www.opensolaris.org/os/community/performance/filebench`, 2006.

[25] Microsoft. Distributed File System (DFS). `http://www.microsoft.com/windowsserver2003/technologies/storage/dfs/default.mspx`.

[26] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 305–313, San Fransisco, CA, USA, 1992.

[27] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[28] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3: Design and Implementation. In *USENIX Summer*, pages 137–152, 1994.

[29] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS Version 4 Protocol. In *Proceedings of Second International System Administration and Networking (SANE) Conference*, May 2000.

[30] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. A 64-bit, Shared Disk File System for Linux. In *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems*, pages 22–41, San Diego, CA, 1999. IEEE Computer Society Press.

[31] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, Jan. 2002.

[32] S. Shepler and et al. Network File System (NFS) version 4 Protocol. RFC 3530 `http://www.ietf.org/rfc/rfc3530.txt`.

[33] G. Sivathanu and E. Zadok. A Versatile Persistent Caching Framework for File Systems. Technical Report FSL-05-05, Computer Science Department, Stony Brook University, Dec 2005.

[34] SNIA. Common Internet File System (CIFS) Technical Reference. `http://www.snia.org/tech_activities/CIFS/CIFS-TR-1p00_FINAL.pdf`.

[35] V. Srinivasan and J. Mogul. Spritely nfs: experiments with cache-consistency protocols. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 45–57, Dec. 1989.

[36] H. Stern, M. Eisler, and R. Labiaga. *Managing NFS and NIS*. O'Reilly, Jul 2001.

TFS: A Transparent File System for Contributory Storage

James Cipar Mark D. Corner Emery D. Berger

Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{jcipar, mcorner, emery}@cs.umass.edu

Abstract

Contributory applications allow users to donate unused resources on their personal computers to a shared pool. Applications such as SETI@home, Folding@home, and Freenet are now in wide use and provide a variety of services, including data processing and content distribution. However, while several research projects have proposed contributory applications that support peer-to-peer storage systems, their adoption has been comparatively limited. We believe that a key barrier to the adoption of contributory storage systems is that contributing a large quantity of local storage interferes with the principal user of the machine.

To overcome this barrier, we introduce the Transparent File System (TFS). TFS provides background tasks with large amounts of unreliable storage—all of the currently available space—without impacting the performance of ordinary file access operations. We show that TFS allows a peer-to-peer contributory storage system to provide 40% more storage at twice the performance when compared to a user-space storage mechanism. We analyze the impact of TFS on replication in peer-to-peer storage systems and show that TFS does not appreciably increase the resources needed for file replication.

1 Introduction

Contributory applications allow users to donate unused resources from their personal computers to a shared pool. These applications harvest idle resources such as CPU cycles, memory, network bandwidth, and local storage to serve a common distributed system. These applications are distinct from other peer-to-peer systems because the resources being contributed are not directly consumed by the contributor. For instance, in Freenet [8], all users contribute storage, and any user may make use of the storage, but there is no relationship between user data and contributed storage. Contributory applications in wide use include computing efforts like Folding@home [17] and anonymous publishing and content distribution such as Freenet [8]. The research community has also developed a number of contributory applications, including distributed backup and archival storage [30], server-less network file systems [1], and distributed web caching [11]. However, the adoption of storage-based contributory applications has been limited compared to those that are CPU-based.

Two major barriers impede broader participation in contributory storage systems. First, existing contributory storage systems degrade normal application performance. While *transparency*—the effect that system performance is as if no contributory application is running—has been the goal of other OS mechanisms for network bandwidth [34], main memory [7], and disk scheduling [19], previous work on contributory storage systems has ignored its local performance impact. In particular, as more storage is allocated, the performance of the user's file system operations quickly degrades [20].

Second, despite the fact that end-user hard drives are often half empty [10, 16], users are generally reluctant to relinquish their free space. Though disk capacity has been steadily increasing for many years, users view storage space as a limited resource. For example, three of the Freenet FAQs express the implicit desire to donate less disk space [12]. Even when users are given the choice to limit the amount of storage contribution, this option requires the user to decide *a priori* what is a reasonable contribution. Users may also try to donate as little as possible while still taking advantage of the services provided by the contributory application, thus limiting its overall effectiveness.

Contributions: This paper presents the Transparent File System (TFS), a file system that can contribute 100% of the idle space on a disk while imposing a negligible performance penalty on the local user. TFS operates by storing files in the free space of the file system so that they are invisible to ordinary files. In essence, normal file allocation proceeds as if the system were not contributing any space at all. We show in Section 5 that TFS imposes nearly no overhead on the local user. TFS achieves this both by minimizing interference with the file system's block allocation policy and by sacrificing persistence for contributed space: normal files may overwrite contributed space at any time. TFS takes several steps that limit this unreliability, but because contributory applications are already designed to work with unreliable machines, they behave appropriately in the face of unreliable files. Furthermore, we show that TFS does not appreciably impact the bandwidth needed for replication. Users typically create little data in the course of a day [4], thus the erasure of contributed storage is negligible when compared to the rate of machine failures.

TFS is especially useful for replicated storage systems executing across relatively stable machines with plentiful bandwidth, as in a university or corporate network. This environment is the same one targeted by distributed storage systems such as FARSITE [1]. As others have shown previously, for high-failure modes, such as wide-area Internet-based systems, the key limitation is the bandwidth between nodes, not the total storage. The bandwidth needed to replicate data after failures essentially limits the amount of storage the network can use [3]. In a stable network, TFS offers substantially more storage than dynamic, user-space techniques for contributing storage.

Organization: In Section 2, we first provide a detailed explanation of the interference caused by contributory applications, and discuss current alternatives for contributing storage. Second, we present the design of TFS in Section 3, focusing on providing transparency to normal file access. We describe a fully operating implementation of TFS. We then explain in Section 4 the interaction between machine reliability, contributed space, and the amount of storage used by a contributory storage system. Finally, we demonstrate in Section 5 that the performance of our TFS prototype is on par with the file system it was derived from, and up to twice as fast as user-space techniques for contributing storage.

2 Interference from Contributing Storage

All contributory applications we are aware of are configured to contribute a small, fixed amount of storage—the contribution is small so as not to interfere with normal machine use. This low level of contribution has little impact on file system performance and files will generally only be deleted by the contributory system, not because the user needs storage space. However, such small, fixed-size contributions limit contribution to small-scale storage systems.

Instead of using static limits, one could use a dynamic system that monitors the amount of storage used by local applications. The contributory storage system could then use a significantly greater portion of the disk, while yielding space to the local user as needed. Possible approaches include the watermarking schemes found in Elastic Quotas [18] and FS^2 [16]. A contributory storage system could use these approaches as follows: whenever the current allocation exceeds the maximum watermark set by the dynamic contribution system, it could delete contributory files until the contribution level falls below a lower watermark.

However, if the watermarks are set to comprise all free space on the disk, the file system is forced to delete files synchronously from contributed storage when writing new files to disk. In this case, the performance of the disk would be severely degraded, similar to the synchronous cleaning problem in LFS [31]. For this reason, Elastic Quotas and FS^2 use more conservative watermarks (e.g., at most 85%), allowing the system to delete files lazily as needed.

Choosing a proper watermark leaves the system designer with a trade-off between the amount of storage contributed and local performance. At one end of the spectrum, the system can contribute little space, limiting its usefulness. At the other end of the spectrum, local performance suffers.

To see why local performance suffers, consider the following: as a disk fills, the file system's block allocation algorithm becomes unable to make ideal allocation decisions, causing fragmentation of the free space and allocated files. This fragmentation increases the seek time when reading and writing files, and has a noticeable effect on the performance of disk-bound processes. In an FFS file system, throughput can drop by as much as 77% in a file system that is only 75% full versus an empty file system [32]—the more storage one contributes, the worse the problem becomes. The only way to avoid this is to maintain enough free space on the disk to allow the allocation algorithm to work properly, but this limits contribution to only a small portion of the disk.

Though some file systems provide utilities to defragment their disk layout, these utilities are ineffective when there is insufficient free space on the file system. For instance, the defragmentation utility provided with older versions of Microsoft Windows will not even attempt to defragment a disk if more than 85% is in use. On modern Windows systems, the defragmentation utility will run when the disk is more than 85% full, but will give a warning that there is not enough free space to defragment properly [22]. When one wants to contribute all of the free space on the disk, they will be unable to meet these requirements of the defragmentation utility.

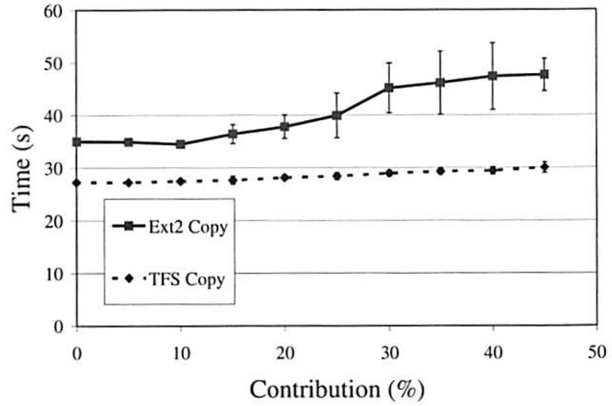


Figure 1: The time required to perform a series of file system operations while contributing different amounts of storage. As the amount of contributed space increases, the time it takes for Ext2 to complete the experiment also increases. However, the performance of TFS stays nearly constant. Error bars represent standard deviation.

Any user-space scheme to manage contributed storage will be plagued by the performance problems introduced by the contributed storage—filling the disk with data inevitably slows access. Given a standard file system interface, it is simply not possible to order the allocation of data on disk to preserve performance for normal files. As Section 3 shows, by incorporating the mechanisms for contribution into the file system itself, TFS maintains file system performance even at high levels of contribution.

Figure 1 depicts this effect. This figure shows the time taken to run the copy phase of the Andrew Benchmark on file systems with different amounts of space being consumed. The full details of the benchmark are presented in Section 5. As the amount of consumed space increases, the time it takes to complete the benchmark increases. We assume that the user is using 50% of the disk space for non-contributory applications, which corresponds to results from a survey of desktop file system contents [10]. The figure shows that contributing more than 20% of the disk space will noticably affect the file system's performance, even if the contributed storage would have been completely idle. As a preview of TFS performance, note that when contributing 35% of the disk, TFS is *twice as fast* as Ext2 for copying files.

3 Design and Implementation of TFS

The Transparent File System (TFS) allows users to donate storage to distributed storage systems with minimal performance impact. Because the block allocation policy is one of the primary determinants of file system performance, designers have devoted considerable attention to tuning it. Accordingly, deviating from that policy can result in a loss of performance. The presence of data on the file system can be viewed as an obstruction which causes a deviation from the default allocation policy. The goal of TFS is to ensure the *transparency* of contributed data: the presence of contributed storage should have no measurable effect on the file system, either in performance, or in capacity. We use the term *transparent files* for files which have this property, and *transparent data* or *transparent blocks* for the data belonging to such files. A *transparent application* is an application which strives to minimize its impact on other applications running on the same machine, possibly at the expense of its own performance.

Section 3.1 shows how we achieve transparency with respect to block allocation in the context of a popular file system, Ext2 [5]. Ext2 organizes data on disk using several rules of thumb that group data on disk according to logical relationships. As we show in Section 5, TFS minimally perturbs the allocation policy for ordinary files, yielding a near-constant ordinary file performance regardless of the amount of contributed storage.

In exchange for this performance, TFS sacrifices file persistence. When TFS allocates a block for an ordinary file, it treats free blocks and transparent blocks the same, and thus may overwrite transparent data. Files marked transparent may be overwritten and deleted at any time. This approach may seem draconian, but because replicated systems already deal with the failure of hosts in the network, they can easily deal with the loss of individual files. For instance, if one deletes a file from a BitTorrent peer, other peers automatically search for hosts that have the file.

The design of TFS is centered around tracking which blocks are allocated to which kind of file, preserving persistence for normal user files, and detecting the overwriting of files in the contributed space. As the file system is now allowed to overwrite certain other files, it is imperative that it not provide corrupt data to the contribution system, or worse yet, to the user. While our design can preserve transparency, we have also made several small performance concessions which have minimal effect on normal file use, but yield a better performing contribution system. Additionally, file systems inevitably have hot spots, possibly leading to continuous allocation and deallocation of space to and from contribution. These hot spots could lead to increased replication traffic elsewhere in the network. TFS incorporates a mechanism that detects these hot-spots and avoids using them for contribution.

3.1 Block Allocation

TFS ensures good file system performance by minimizing the amount of work that the file system performs when writing ordinary files. TFS simply treats transparent blocks as if they were free, overwriting whatever data might be currently stored in them. This policy allows block allocation for ordinary files to proceed exactly as it would if there were no transparent files present. This approach preserves performance for ordinary files, but corrupts data stored in transparent files. If an application were to read the transparent file after a block was overwritten, it would receive the data from the ordinary file in place of the data that had been overwritten. This presents two issues: applications using transparent files must ensure the correctness of all file data, and sensitive information stored in ordinary files must be protected from applications trying to read transparent files. To prevent both effects, TFS records which blocks have been overwritten so that it can avoid treating the data in those blocks as valid transparent file data. When TFS overwrites a transparent file, it marks it as *overwritten* and allocates the block to the ordinary file.

This requires some modifications to the allocation policy in Ext2. Blocks in a typical file system can only be in one of two states: *free* and *allocated.* In contrast, in TFS a storage block can be in one of five states: *free, allocated, transparent, free-and-overwritten,* and *allocated-and-overwritten.*

Figure 2 shows a state transition diagram for TFS blocks. Ordinary data can be written over free or transparent blocks. If the block was previously used by transparent data, the file system marks these blocks as allocated-and-overwritten. When a block is denoted as overwritten, it means that the transparent data has been overwritten, and thus "corrupted" at some point. Transparent data can only be written to free blocks. It cannot overwrite allocated blocks, other transparent blocks, or overwritten blocks of any sort. Figure 3 shows this from the perspective of the block map. Without TFS, appending to a file leads to fragmentation, leading to a performance loss in the write, and in subsequent reads. In TFS, the file remains contiguous, but the transparent file data is lost and the blocks are marked as *allocated-and-overwritten.*

When a process opens a transparent file, it must verify that none of the blocks have been overwritten since the last time it was opened. If any part of the file is overwritten, the file system returns an error to `open`. This signals that the file has been deleted. TFS then deletes the inode and directory entry for the file, and marks all of the blocks of the file as *free*, or *allocated.* As ordinary files cannot ever be overwritten, scanning the allocation bitmaps is not necessary when opening them. This lazy-delete scheme means that if TFS writes transparent files and never uses them again, the disk will eventually fill with overwritten blocks that could otherwise be used by the transparent storage application. To solve this, TFS employs a simple, user-space cleaner that opens and closes transparent files on disk. Any corrupted files will be detected and automatically deleted by the `open` operation.

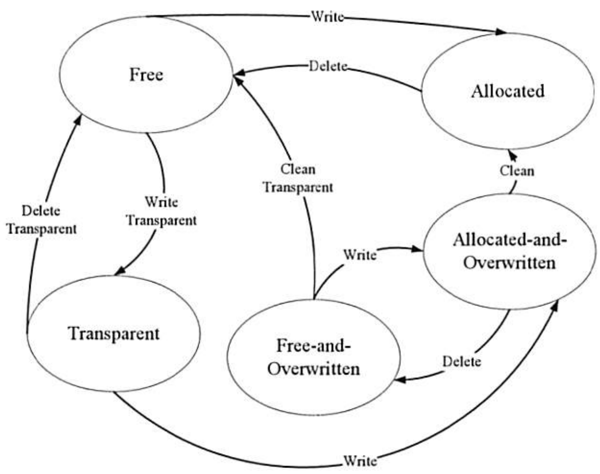


Figure 2: A state diagram for block allocation in TFS. The *Free* and *Allocated* states are the two allocation states present in the original file system. TFS adds three more states.

Though scanning the blocks is a linear operation in the size of the file, very little data must be read from the disk to scan even very large files. On a typical Ext2 file system, if we assume that file blocks are allocated contiguously, then scanning the blocks for a 4GB file will only require 384kB to be read from the disk. In the worst case—where the file is fragmented across *every* block group, and we must read every block bitmap—approximately 9.3MB will be read during the scan, assuming a 100GB disk.

Many file systems, including Ext2 and NTFS, denote a block's status using a bitmap. TFS augments this bitmap with two additional bitmaps and provides a total of three bits denoting one of the five states. In a 100GB file system with 4kB blocks, these bitmaps use only 6.25MB of additional disk space. These additional bitmaps must also be read into memory when manipulating files. However, very little of the disk will be actively manipulated at any one time, so the additional memory requirements are negligible.

3.2 Performance Concessions

This design leads to two issues: how TFS deals with open transparent files and how TFS stores transparent metadata. In each case, we make a small concession to transparent storage at the expense of ordinary file system performance. While both concessions are strictly unnecessary, their negative impact on performance is negligible and their positive impact on transparent performance is substantial.

First, as TFS verifies that all blocks are clean only at open time, it prevents the file system from overwriting the data of open transparent files. One alternative would be to close the transparent file and kill the process with the open file descriptor if an overwritten block is detected. However, not only would it be difficult to trace blocks to file descriptors, but it could also lead to data corruption in the transparent process. In our opinion, yielding to open files is the best option. We discuss other strategies in Section 7.

It would also be possible to preserve the transparent data by implementing a copy-on-write scheme [26]. In this case, the ordinary file block would still allocate its target, and the transparent data block would be moved to another location. This is to ensure transparency with regards to the ordinary file allocation policy. However, to use this strategy, there must be a way to efficiently determine which inode the transparent data block belongs to, so that it can be relinked to point to the new block. In Ext2, and most other file systems, the file system does not keep a mapping from data blocks to inodes. Accordingly, using copy-on-write to preserve transparent data would require a scan of all inodes to determine the owner of the block, which would be prohibitively expensive. It is imaginable that a future file system would provide an efficient mapping from data blocks to inodes, which would allow TFS to make use of copy-on-write to preserve transparent data, but this conflicts with our goal of requiring minimal modifications to an existing operating system.

Second, TFS stores transparent meta-data such as inodes and indirect blocks as ordinary data, rather than transparent blocks. This will impact the usable space for ordinary files and cause some variation in ordinary block allocation decisions. However, consider what would happen if the transparent meta-data were overwritten. If the data included the root inode of a large amount of transparent data, all of that data would be lost and leave an even larger number of garbage blocks in the file system. Determining liveness typically requires a full tracing from the root as data blocks do not have reverse mappings to inodes and indirect blocks. Storing transparent storage metadata as ordinary blocks avoids both issues.

3.3 Transparent Data Allocation

As donated storage is constantly being overwritten by ordinary data, one concern is that constant deletion will have ill effects on any distributed storage system. Every time a file is deleted, the distributed system must detect and replicate that file, meanwhile returning errors to any peers that request it. To mitigate these effects, TFS identifies and avoids using the hot spots in the file system that could otherwise be used for donation. The total amount of space that is not used for donation depends on the bandwidth limits of the distributed system and is configurable, as shown in this section.

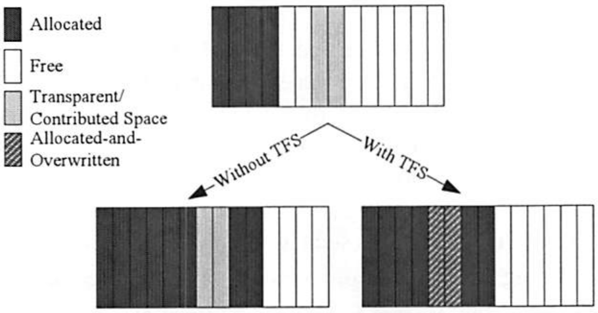


Figure 3: The block map in a system with and without TFS. When a file is appended in a normal file system it causes fragmentation, while in TFS, it yields two overwritten blocks.

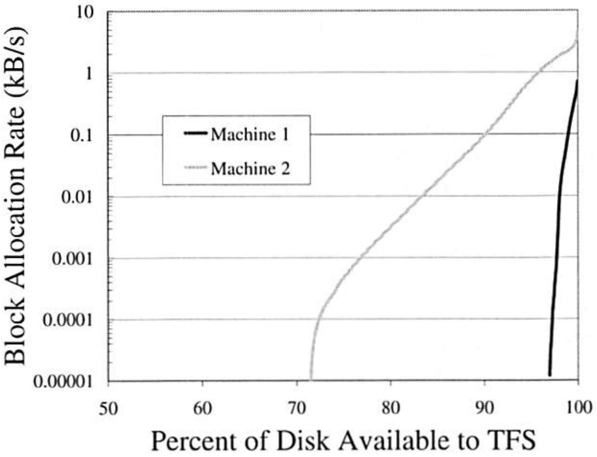


Figure 4: Cumulative histogram of two user machine's block allocations. 70% of the blocks on machine 2's disk were never allocated during the test period, and 90% of the blocks were allocated at a rate of 0.1kB/s or less. This corresponds to the rate at which TFS would overwrite transparent data if it were to use a given amount of the disk.

By design, the allocation policy for Ext2 and other logically organized file systems exhibits a high degree of spatial locality. Blocks tend to be allocated to only a small number of places on the disk, and are allocated repeatedly. To measure this effect, we modified a Linux kernel to record block allocations on two user workstations machines in our lab. A cumulative histogram of the two traces is shown in Figure 4. Machine 1 includes 27 trace days, and machine 2 includes 13 trace days. We can observe two behaviors from this graph. First, while one user is a lot more active than the other, both show a great deal of locality in their access—machine 2 never allocated any blocks in 70% of the disk. Second, an average of 1kB/s of block allocations is approximately 84MB of allocations per day. Note that this is not the same as creating 84MB/s of data per day—the trace includes many short-lived allocations such as temporary lock files.

Using this observation as a starting point, TFS can balance the rate of block deletion with the usable storage on the disk. Using the same mechanism that we used to record the block allocation traces shown in Figure 4, TFS generates a trace of the block addresses of all ordinary file allocations. It maintains a histogram of the number of allocations that occurred in each block and periodically sorts the blocks by the number of allocations. Using this sorted list, it finds the smallest set of blocks responsible for a given fraction of the allocations.

The fraction of the allocations to avoid, f, affects the rate at which transparent data is overwritten. Increasing the value of f means fewer ordinary data allocations will overwrite transparent data. On the other hand, by decreasing the value of f, more storage becomes available to transparent data. Because the effects of changing f are dependent on a particular file system's usage pattern, we have found it convenient to set a target loss rate and allow TFS to determine automatically an appropriate value for f. Suppose ordinary data blocks are allocated at a rate of α blocks per second. If f is set to 0 – meaning that TFS determines that the entire disk is usable by transparent data – then transparent data will be overwritten at a rate approximately equal to α. The rate at which transparent data is overwritten t is approximately $\beta = (1 - f)\alpha$. Solving for f gives $f = 1 - \frac{\beta}{\alpha}$. Using this, TFS can determine the amount of storage available to transparent files, given a target rate. Using this map of hot blocks in the file system, the allocator for transparent blocks treats them as if they were already allocated to transparent data.

However, rather than tracking allocations block-by-block, we divide the disk into groups of disk blocks, called chunks, and track allocations to chunks. Each chunk is defined to be one eighth of a block group. This number was chosen so that each block group could keep a single byte as a bitmap representing which blocks should be avoided by transparent data. For the default block size of 4096 bytes, and the maximum block group size, each chunk would be 16MB.

Dividing the disk into multi-block chunks rather than considering blocks individually greatly reduces the memory and CPU requirements of maintaining the histogram, and due to spatial locality, a write to one block is a good predictor of writes to other blocks in the same chunk. This gives the histogram predictive power in avoiding hot blocks.

It should be noted that the rate at which transparent data is overwritten is not exactly α because, when a transparent data block is overwritten, an entire file is lost. However, because of the large chunk size and the high locality of chunk allocations, subsequent allocations for ordinary data tend to overwrite other blocks of the same transparent file, making the rate at which transparent data lost approximately equal to the rate at which blocks are overwritten.

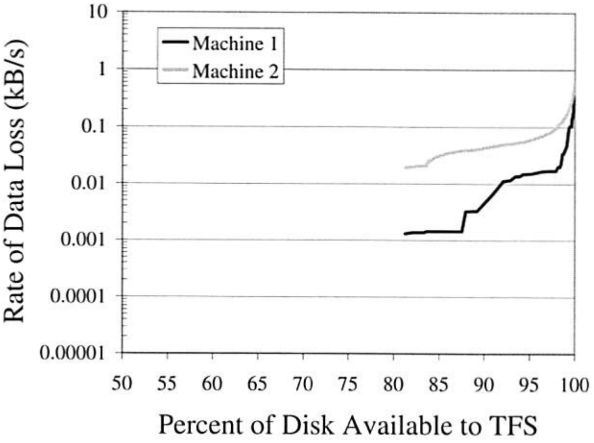


Figure 5: This shows the simulated rate of TFS data loss when using block avoidance to avoid hot spots on the disk. For example, when TFS allows 3% of the disk to go unused, the file system will allocate data in the used portion of the disk at a rate of 0.1kB/s. By using block avoidance, TFS provides more storage to contributory applications without increasing the rate at which data is lost.

The figure of merit is the rate of data erased for a reasonably high allocation of donated storage. To examine TFS under various allocations, we constructed a simulator using the block allocation traces used earlier in the section. The simulator processes the trace sequentially, and periodically picks a set of chunks to avoid. Whenever the simulator sees an allocation to a chunk which is not being avoided, it counts this as an overwrite. We ran this simulator with various fixed values of f, the fraction of blocks to avoid, and recorded the average amount of space contributed, and the rate of data being overwritten. Figure 5 graphs these results. Note that the graph starts at 80% utilization. This is dependent on the highest value for f we used in our test. In our simulation, this was 0.99999. Also notice that, for contributions less than approximately 85%, the simulated number of overwrites is greater than the number given in Figure 4. This is because, for very high values of f, the simulator's adaptive algorithm must choose between many chunks, none of which have received very many allocations. In this case, it is prone to make mistakes, and may place transparent data in places that will see allocations in the near future. These results demonstrate that for machine 2's usage pattern, TFS can donate all but 3% of the disk, while only erasing contributed storage files at 0.08kB/s. As we demonstrate in the section 4, when compared to the replication traffic due to machine failures, this is negligible.

3.4 TFS Implementation

We have implemented a working prototype of TFS in the Linux kernel (2.6.13.4). Various versions of TFS have been used by one of the developers for over six months to store his home directory as ordinary data and Freenet data as transparent data.

TFS comprises an in-kernel file system and a user-space tool for designating files and directories as either transparent or opaque, called `setpri`. We implemented TFS using Ext2 as a starting point, adding or modifying about 600 lines of kernel code, in addition to approximately 300 lines of user-space code. The primary modifications we made to Ext2 were to augment the file system with additional bitmaps, and to change the block allocation to account for the states described in Section 3. Additionally, the `open` VFS call implements the lazy-delete system described in the design. In user-space, we modified several of the standard tools (including `mke2fs` and `fsck`) to use the additional bitmaps that TFS requires. We implemented the hot block avoidance histograms in user-space using a special interface to the kernel driver. This made implementation and experimentation somewhat easier; however, future versions will incorporate those functions into the kernel. An additional benefit is that the file system reports the amount of space available to ordinary files as the free space of this disk. This causes utilities such as `df`, which are used to determine disk utilization, to ignore transparent data. This addresses the concerns of users who may be worried that contributory applications are consuming their entire disk.

In our current implementation, the additional block bitmaps are stored next to the original bitmaps as file system metadata. This means that our implementation is not backwards-compatible with Ext2. However, if we moved the block bitmaps to Ext2 data blocks, we could create a completely backwards-compatible version, easing adoption. We believe that TFS could be incorporated into almost any file system, including Ext3 and NTFS.

4 Storage Capacity and Bandwidth

The usefulness of TFS depends on the characteristics of the distributed system it contributes to, including the dynamics of machine availability, the available bandwidth, and the quantity of available storage at each host. In this section, we show the relationship between these factors, and how they affect the amount of storage available to contributory systems. We define the storage contributed as a function of the available bandwidth, the uptime of hosts, and the rate at which hosts join and leave the network. Throughout the section we will be deriving equations which will be used in our evaluation.

4.1 Replication Degree

Many contributory storage systems use replication to ensure availability. However, replication limits the capacity of the storage system in two ways. First, by storing redundant copies of data on the network, there is less overall space [2]. Second, whenever data is lost, the system must create a new replica.

First we calculate the degree replication needed as a function of the average node uptime. We assume that nodes join and leave the network independently. Though this assumption is not true in the general case, it greatly simplifies the calculations here, and holds true in networks where the only downtime is a result of node failures, such as a corporate LAN. In a WAN where this assumption does not hold, these results still provide an approximation which can be used as insight towards the system's behavior. Mickens and Noble provide a more in-depth evaluation of availability in peer-to-peer networks [21].

To determine the number of replicas needed, we use a result from Blake and Rodrigues [3]. If the fraction of time each host was online is u, and each file is replicated r times, then the probability that no replicas of a file will be available at a particular time is

$$(1-u)^r. \tag{1}$$

To maintain an availability of a, the number of replicas must satisfy the equation

$$a = 1 - (1-u)^r. \tag{2}$$

Solving for r gives the number of replicas needed.

$$r = \frac{ln(1-a)}{ln(1-u)} \tag{3}$$

We consider the desired availability a to be a fixed constant. A common rule of thumb is that "five nines" of availability, or $a = 0.99999$ is acceptable, and the value u is a characteristic of host uptime and downtime in the network. Replication could be accomplished by keeping complete copies of each file, in which case r would have to be an integer. Replication could also be implemented using a coding scheme that would allow non-integer values for r [28], and a different calculation for availability. In our analysis, we simply assume that r can take any value greater than 1.

4.2 Calculating the Replication Bandwidth

The second limiting effect in storage is the demand for replication bandwidth. As many contributory systems exhibit a high degree of *churn*, the effect of hosts frequently joining and leaving the network [13, 33], repairing failures can prevent a system from using all of its available storage [3]. When a host leaves the network for any reason, it is unknown when or if the host will return. Accordingly, all files which the host was storing must be replicated to another machine. For hosts that were storing a large volume of data, failure imposes a large bandwidth demand on the remaining machines. For instance, a failure of one host storing 100GB of data every 100 seconds imposes an aggregate bandwidth demand of 1GB/s across the remaining hosts. In this section, we consider the average bandwidth consumed by each node. When a node leaves the network, all of its data must be replicated. However, this data does not have to be replicated to a single node. By distributing the replication, the maximum bandwidth demanded by the network can be very close to the average.

The critical metric in determining the bandwidth required for a particular storage size is the *session time* of hosts in the network: the period starting when the host joins the network, and ending when its data must be replicated. This is not necessarily the same as the time a host is online—hosts frequently leave the network for a short interval before returning.

Suppose the average storage contribution of each host is c, and the average session time is T. During a host's session, it must download all of the data that it will store from other machines on the network. With a session time of T, and a storage contribution of c, the average downstream bandwidth used by the host is $B = \frac{c}{T}$. Because all data transfers occur within the contributory storage network, the average downstream bandwidth equals the average upstream bandwidth.

In addition to replication due to machine failures, both TFS and dynamic watermarking cause an additional burden due to the local erasure of files. If each host loses file data at a rate of F, then the total bandwidth needed for replication is

$$B = \frac{c}{T} + F. \tag{4}$$

Solving for the storage capacity as a function of bandwidth gives

$$c = T \cdot (B - F). \tag{5}$$

The file failure rate F in TFS is measurable using the methods of the previous section. The rate at which files are lost when contributing storage by the dynamic watermarking scheme is less than the rate of file loss with TFS. When using watermarking, this rate is directly tied to the rate at which the user creates new data.

If the value c is the total amount of storage contributed by each host in the network, then for a replication factor of r, the amount of unique storage contributed by each host is

$$C = \frac{c}{r} = \frac{T \cdot (B - F)}{r}. \tag{6}$$

The session time, T, is the time between when a host comes online and when its data must be replicated to another host, because it is going offline, or has been offline for a certain amount of time. By employing *lazy replication*—waiting for some threshold of time, t, before replicating its data—we can extend the average session time of the hosts [2]. However, lazy replication reduces the number of replicas of a file that are actually online at any given time, and thus increases the number of replicas needed to maintain availability. Thus, both T, the session time, and r, the degree of replication, are functions of t, this threshold time.

$$C = \frac{T(t)(B - F)}{r(t)} \tag{7}$$

The functions $T(t)$ and $r(t)$ are sensitive to the failure model of the network in question. For instance, in a corporate network, machine failures are rare, and session times are long. However, in an Internet-based contributory system, users frequently sign on for only a few hours at time.

5 TFS Evaluation

Our goal in evaluating TFS is to assess its utility for contributing storage to a peer-to-peer file system. We compare each method of storage contribution that we describe in Section 2 to determine how much storage can be contributed, and the effects on the user's application performance. We compare these based on several metrics: the amount of storage contributed, the effect on the block allocation policy, and the overall effect on local performance.

In scenarios that are highly dynamic and bandwidth-limited, static contribution yields as much storage capacity as any of the other three. If the network is more stable, and has more bandwidth, the dynamic scheme provides many times the storage of the static scheme; however, it does so at the detriment of local performance. When bandwidth is sufficient and the network is relatively stable, as in a corporate network, TFS provides 40% more storage than dynamic watermarking, with no impact on local performance. TFS always provides at least as much storage as the other schemes without impacting local performance.

5.1 Contributed Storage Capacity

To determine the amount of storage available to a contributory system, we conduct trace-based analyses using the block avoidance results from Section 3, the analysis of the availability trace outlined in Section 3.3, and the relationship between bandwidth and storage described in Section 4.2. From these, we use Equation 7 to determine the amount of storage that can be donated by each host in a network using each of the three methods of contribution.

We assume a network of identical hosts, each with 100GB disks that are 50% full, and we use the block traces for Machine 2 in Section 3.3 as this machine represents the worst-case of the two machines. Given a fixed rate of data loss caused by TFS, we determine the maximum amount of data that can be stored by TFS based on the data from Figure 5. We assume that the fixed contribution and the dynamic contribution methods cause no data loss. Though the dynamic contribution method does cause data loss as the user creates more data on the disk, the rate of data creation by users in the long term is low [4]. We assume that the amount of non-contributed storage being used on the disk is fixed at 50%. For fixed contribution, each host contributes 5% of the disk (5 GB), the dynamic system contributes 35% of the disk, and TFS contributes about 47%, leaving 3% free to account for block avoidance.

To determine the functions $T(t)$ and $r(t)$, we analyze availability traces gathered from two different types of networks which exhibit different failure behavior. These networks are the Microsoft corporate network [4] and Skype super-peers [14]. The traces contain a list of time intervals for which each host was online contiguously. To determine the session time for a given threshold t, we first combine intervals that were separated by less than t. We then use the average length of the remaining intervals and add the value t to it. The additional period t represents the time after a node leaves the network, but before the system decides to replicate its data.

We use these assumptions to calculate the amount of storage given to contributory systems with different amounts of bandwidth. For each amount of bandwidth, we find the value for the threshold time (Section 4.2) that maximizes the contributed storage for each combination of file system, availability trace, and bandwidth. We use this value to compute the amount of storage available using Equation 7. Figure 6 shows the bandwidth vs. storage curve using the reliability model based on availability traces of corporate workstations at Microsoft [4]. Figure 7 shows similar curves using the reliability model derived from traces of the Skype peer-to-peer Internet telephone network [14].

Each curve has two regions. In the first region, the total amount of storage is limited by the available bandwidth, and increases linearly as the bandwidth is increased. The slope of the first part of the curve is determined by the frequency of machine failures and file failures. This line is steeper in networks where hosts tend to be more reliable, because less bandwidth is needed to replicate a large amount of data. In this case, the amount of available storage does not affect the amount of usable storage. This means that, for the first part of the curve when the systems are bandwidth-limited, TFS contributes an amount of storage similar to the other two systems. Though TFS contributes slightly less because of file failures, the additional bandwidth needed to handle failures is small.

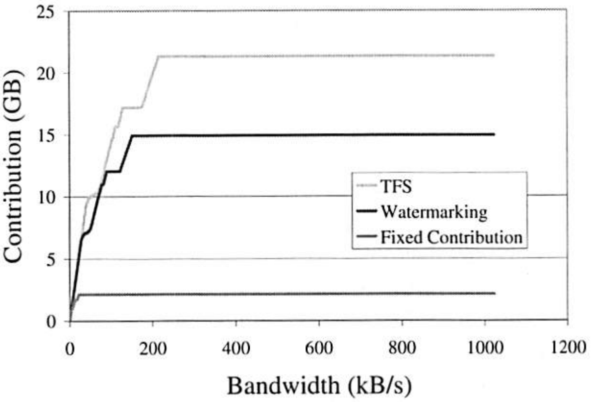


Figure 6: The amount of storage that can be donated for contributions of network bandwidth, assuming a corporate-like failure model. With reliable machines, TFS is able to contribute more storage than other systems, even when highly bandwidth-limited.

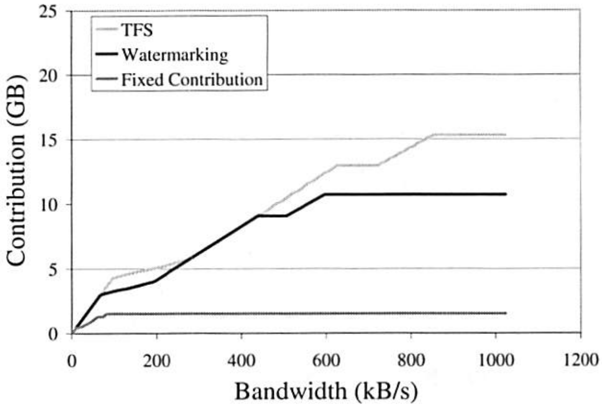


Figure 7: The amount of storage that can be donated for contributions of network bandwidth, assuming an Internet-like failure model. Because peer-to-peer nodes on the Internet are less reliable, the amount of contribution by TFS does not surpass the other techniques until large amounts of bandwidth become available.

The second part of the curve starts when the amount of storage reaches the maximum allowed by that contribution technique. For instance, when the small contribution reaches 5% of the disk, it flattens out. This part of the curve represents systems that have sufficient replication bandwidth to use all of the storage they are given, and are only limited by the amount of available storage. In this case, TFS is capable of contributing significantly more storage than other methods.

In the Microsoft trace, the corporate systems have a relatively high degree of reliability, so the bandwidth-limited portion of the curves is short. This high reliability means that, even for small bandwidth allocations, TFS is able to contribute the most storage. The Skype system shows a less reliable network of hosts. Much more network bandwidth is required before TFS is able to contribute more storage than the other storage techniques can—in fact, much more bandwidth than is typically available in Internet connected hosts. However, even when operating in a bandwidth-limited setting, TFS is able to contribute as much as the other techniques. One method to mitigate these bandwidth demands is to employ background file transfer techniques such as TCP-Nice [34].

From these results, we can conclude that TFS donates nearly as much storage as other methods in the worst case. However, TFS is most effective for networks of reliable machines, where it contributes 40% more storage than a dynamic watermarking system. It is important to note that these systems do not exhibit the same impact on *local* performance, which we demonstrate next.

5.2 Local File System Performance

To show the effects of each system on the user's file system performance, we conduct two similar experiments. In the first experiment, a disk is filled to 50% with ordinary file data. To achieve a realistic mix of file sizes, these files were taken from the `/usr` directory on a desktop workstation. These files represent the user's data and do not change during the course of the experiment. After this, files are added to the system to represent the contributed storage.

We considered four cases: no contribution, small contribution, large contribution, and TFS. The case where there is no simulated contribution is the baseline. Any decrease in performance from this baseline is interference caused by the contributed storage. The small contribution is 5% of the file system. This represents a fixed contribution where the amount of storage contributed must be set very small. The large contribution is 35% of the file system. This represents the case of dynamically managed contribution, where a large amount of storage can be donated. With TFS, the disk is filled completely with transparent data. Once the contributed storage is added, we run a version of the Modified Andrew Benchmark [25] to determine the contribution's effect on performance.

We perform all of the experiments using two identical Dell Optiplex SX280 systems with an Intel Pentium 4 3.4GHz CPU, 800MHz front side bus, 512MB of RAM, and a 160GB SATA 7200RPM disk with 8MB of cache. The trials were striped across the machines to account for any subtle differences in the hardware. We conduct ten trials of each experiment, rebooting between each trial, and present the average of the results. The error bars in all figures in this section represent the standard deviation of our measurements. In all cases, the standard deviation was less than 14% of the mean.

5.2.1 The Andrew Benchmark

The Andrew Benchmark [15] is designed to simulate the workload of a development workstation. Though most users do not compile programs frequently, the Andrew Benchmark can be viewed as a test of general small-file performance, which is relevant to all workstation file systems. The benchmark starts with a source tree located on the file system being tested. It proceeds in six phases: mkdir, copy, stat, read, compile, and delete.

Mkdir During the mkdir phase, the directory structure of the source tree is scanned, and recreated in another location on the file system being tested.

Copy The copy phase then copies all of the non-directory files from the original source tree to the newly created directory structure. This tests small file performance of the target file system, both in reading and writing.

Stat The stat phase then scans the newly created source tree and calls `stat` on every file.

Read The read phase simply reads all data created during the copy phase.

Compile The compile phase compiles the target program from the newly created source tree.

Delete The delete phase deletes the new source tree.

The Andrew Benchmark has been criticized for being an old benchmark, with results that are not meaningful to modern systems. It is argued that the workload being tested is not realistic for most users. Furthermore, the original Andrew Benchmark used a source tree which is too small to produce meaningful results on modern systems [15]. However, as we stated above, the Benchmark's emphasis on small file performance is still relevant to modern systems. We modified the Andrew Benchmark to use a Linux 2.6.14 source tree, which consists of 249MB of data in 19577 files. Unfortunately, even with this larger source tree, most of the data used by the benchmark can be kept in the operating system's page cache. The only phase where file system performance has a significant impact is the copy phase.

Despite these shortcomings, we have found that the results of the Andrew Benchmark clearly demonstrate the negative effects of contributing storage. Though the only significant difference between the contributing case and the non-contributing case is in the copy phase of the benchmark, we include all results for completeness.

The results of this first experiment are shown in Figure 8. The only system in which contribution causes any appreciable effect on the user's performance is the case of a large contribution with Ext2. Both the small contribution and TFS are nearly equal in performance to the case of no contribution.

It is interesting to note that the performance of TFS with 50% contribution is slightly better than the performance of Ext2 with 0% contribution. However, this does not show that TFS is generally faster than Ext2, but that for this particular benchmark TFS displays better performance. We suspect that this is an artifact of the way the block allocation strategy was be modified to accommodate TFS. As we show in Section 5.3, when running our Andrew Benchmark experiment, TFS tends to allocate the files used by the benchmark to a different part of the disk than Ext2, which gives TFS slightly better performance compared to Ext2. This is not indicative of a major change in the allocation strategy; Ext2 tries to allocate data blocks to the same block group as the inode they belong to [5]. In our Andrew Benchmark experiments, there are already many inodes owned by the transparent files, so the benchmark files are allocated to different inodes, and therefore different block groups.

The second experiment is designed to show the effects of file system aging [32] using these three systems. Smith and Seltzer have noted that aging effects can change the results of file system benchmarks, and that aged file systems provide a more realistic testbed for file system performance. Though our aging techniques are purely artificial, they capture the long term effects of continuously creating and deleting files. As contributory files are created and deleted, they are replaced by files which are often allocated to different places on the disk. The long term effect is that the free space of the disk becomes fragmented, and this fragmentation interferes with the block allocation algorithm. To simulate this effect, we ran an experiment very similar to the first. However, rather than simply adding files to represent contributed storage, we created and deleted contributory files at random, always staying within 5% of the target disk utilization. After repeating this 200 times, we proceeded to benchmark the file system.

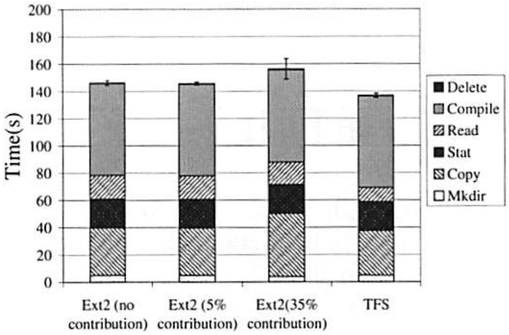


Figure 8: Andrew benchmark results for 4 different unaged file systems. The first is an Ext2 system with no contributory application. The second is Ext2 with a minimal amount of contribution (5%). The third has a significant contribution (35%). The fourth is TFS with complete contribution. TFS performance is on par with Ext2 with no contribution. Error bars represent standard deviation in total time.

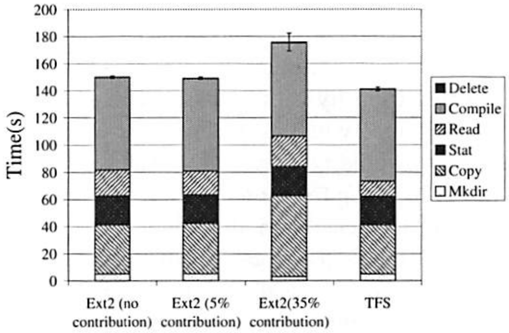


Figure 9: Andrew benchmark results for 4 different aged file systems. The first is an Ext2 system with no contributory application. The second is Ext2 with a minimal amount of contribution (5%). The third has a significant contribution (35%). The fourth is TFS with complete contribution. TFS performance is still comparable to Ext2 with no contribution. Error bars represent standard deviation in total time.

Figure 9 shows the results of this experiment. As with the previous experiment, the only system that causes any interference with the user's applications is the large contribution with Ext2. In this case, Ext2 with 35% contribution takes almost 180 seconds to complete the benchmark. This is about 20 seconds longer than the same system without aging. This shows that file activity caused by contributory applications can have a noticeable impact on performance, even after considering the impact of the presence of those files. On the other hand, the time

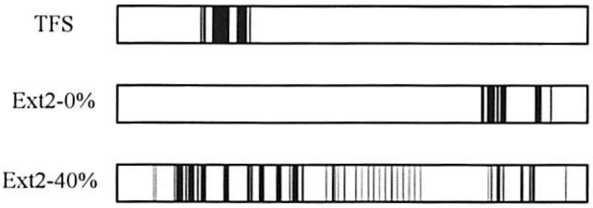


Figure 10: Block allocation pattern for several contribution levels in Ext2 and TFS. Both TFS and Ext2 with 0% contribution show high locality. Ext2 with 40% contribution does not because the contributed files interfere with the block allocation policy.

that TFS takes to complete the benchmark is unchanged by aging in the contributory files. There are no long-term effects of file activity by contributory systems in TFS.

5.3 Block Allocation Layout

A closer look at the block allocation layout reveals the cause of the performance difference between Ext2 and TFS. We analyzed the block layout of files in four occupied file systems. Two of these systems were using Ext2, the third was TFS. Each file system was filled to 50% capacity with ordinary data. Data was then added to simulate different amounts of data contribution. For Ext2, we consider two levels of contribution: 0% and 40%. 0% and 40% were chosen as examples of good and bad Ext2 performance from Figure 1. The TFS system was filled to 50% capacity with ordinary data, and the rest of the disk was filled with transparent files. We then copied the files that would be used in the Andrew Benchmark into these disks, and recorded which data blocks were allocated for the benchmark files.

Figure 10 shows the results of this experiment. The horizontal axis represents the location of the block on the disk. Every block that contains data to be used in the Andrew Benchmark is marked black. The remaining blocks are white, demonstrating the amount of fragmentation in the Andrew Benchmark files. Note that both Ext2 with 0% contribution, and TFS show very little fragmentation. However, the 40% case shows a high degree of fragmentation. This fragmentation is the primary cause of the performance difference between Ext2 with and without contribution in the other benchmarks.

6 Related Work

Our work brings together two areas of research: techniques to make use of the free space in file systems, and the study of churn in peer-to-peer networks.

Using Free Disk Space: Recognizing that the file system on a typical desktop is nearly half-empty, researchers have investigated ways to make use of the extra storage. FS^2 [16] is a file system that uses the extra space on the disk for block-level replication to reduce average seek time. FS^2 dynamically analyzes disk traffic to determine which blocks are frequently used together. It then creates replicas of blocks so that the spatial locality on disk matches the observed temporal locality. FS^2 uses a policy that deletes replicas on-demand as space is needed. We believe that it could benefit from a TFS-like allocation policy, where all replicas except the primary one would be stored as transparent blocks. In this way, the entire disk could be used for block replication.

A number of peer-to-peer storage systems have been proposed that make use of replication and free disk space to provide reliability. These include distributed hash tables such as Chord [24] and Pastry [29], as well as complete file systems like the Chord File System [9], and Past [30].

Churn in Peer-to-Peer Networks: The research community has also been active in studying the dynamic behavior of deployed peer-to-peer networks. Measurements of churn in live systems have been gathered and studied as well. Chu et al. studied the availability of nodes in the Napster and Gnutella networks [6]. The Bamboo DHT was designed as an architecture that can withstand high levels of churn [27]. The Bamboo DHT [23] is particularly concerned with using a minimum amount of "maintenance bandwidth" even under high levels of churn. We believe that these studies give a somewhat pessimistic estimate of the stability of future peer-to-peer networks. As machines become more stable and better connected, remain on continuously, and are pre-installed with stable peer-to-peer applications or middleware, the level of churn will greatly diminish, increasing the value of TFS.

7 Future Work

TFS was designed to require minimal support from contributory applications. The file semantics provided by TFS are no different than any ordinary file system. However, modifying the file semantics would provide opportunities for contributory applications to make better use of free space. For instance, if only a small number of blocks from a large file are overwritten, TFS will delete the entire file. One can imagine an implementation of TFS where the application would be able to recover the non-overwritten blocks. When a file with overwritten blocks is opened, the `open` system call could return a value indicating that the file was successfully opened, but some blocks may have been overwritten. The application can then use `ioctl` calls to determine which blocks have been overwritten. An attempt to read data from an overwritten block will fill the read buffer with zeros.

In the current implementation of TFS, once a transparent file is opened, its blocks cannot be overwritten until the file is closed. This allows applications to assume, as they normally do, that once a file is opened, reads and writes to the file will succeed. However, this means that transparent files may interfere with ordinary file activity. To prevent this, it would be possible to allow data from opened files to be overwritten. If an application attempts to read a block which has been overwritten, the read call could return an error indicating why the block is not available.

Both of these features could improve the service provided by TFS. By allowing applications to recover files that have been partially overwritten, the replication bandwidth needed by systems using TFS is reduced to the rate at which the user creates new data. By allowing open files to be overwritten, transparent applications may keep large files open for extended periods without impacting the performance of other applications. Despite these benefits, one of our design goals for TFS is that it should be usable by unmodified applications. Both of these features would require extensive support from contributory applications, violating this principle.

8 Conclusions

We have presented three methods for contributing disk space in peer-to-peer storage systems. We have included two user-space techniques, and a novel file system, TFS, specifically designed for contributory applications. We have demonstrated that the key benefit of TFS is that it leaves the allocation for local files intact, avoiding issues of fragmentation—TFS stores files such that they are completely transparent to local access. The design of TFS includes modifications to the free bitmaps and a method to avoid hot-spots on the disk.

We evaluated each of the file systems based the amount of contribution and its cost to the local user's performance. We quantified the unreliability of files in TFS and the amount of replication bandwidth that is needed to handle deleted files. We conclude that out of three techniques, TFS consistently provides at least as much storage with no detriment to local performance. When the network is relatively stable and adequate bandwidth is available, TFS provides 40% more storage over the best user-space technique. Further, TFS is completely transparent to the local user, while the user-space technique creates up to a 100% overhead on local performance. We believe that the key to encouraging contribution to peer-to-peer systems is removing the barriers to contribution, which is precisely the aim of TFS.

The source code for TFS is available at `http://prisms.cs.umass.edu/tcsm/`.

Acknowledgments

This material is based upon work supported by the National Science Foundation under CAREER Awards CNS-0447877 and CNS-0347339, and DUE 0416863. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

We would also like to thank our anonymous reviewers for their helpful comments. We would like to give special thanks to Erez Zadok for his careful shepherding.

References

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symposium on Operating System Design and Implementation (OSDI 2002)*. USENIX Association, December 2002.

[2] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey Voelker. Total Recall: System support for automated availability management. In *Proceedings of the First ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, pages 73–86, San Jose, CA, May 2004.

[3] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 1–6, Lihue, Hawaii, May 2003.

[4] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*, pages 34–43, New York, NY, USA, 2000. ACM Press.

[5] Remy Card, Thodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*. Laboratoire MASI — Institut Blaise Pascal and Massachussets Institute of Technology and University of Edinburgh, December 1994.

[6] Jacky Chu, Kevin Labonte, and Brian Neil Levine. Availability and locality measurements of peer-to-peer file systems. In *Proceedings of ITCom: Scala-*

bility and Traffic Control in IP Networks II Conference, July 2002.

[7] James Cipar, Mark D. Corner, and Emery D. Berger. Transparent contribution of memory. In *USENIX Annual Technical Conference (USENIX 2006)*, pages 109–114. USENIX, June 2006.

[8] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[10] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1999)*, pages 59–70, New York, NY, USA, 1999. ACM Press.

[11] Michael J. Freedman, Eric Freudenthal, and David Mazieres. Democratizing content publication with Coral. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.

[12] `http://freenetproject.org/faq.html`.

[13] P. Brighten Goedfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. In *Proc. of ACM SIGCOMM*, 2006.

[14] Saikat Guha, Neil Daswani, and Ravi Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *Proceedings of The 5th International Workshop on Peer-to-Peer Systems (IPTPS '06)*, Santa Barbara, CA, February 2006.

[15] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanana, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[16] Hai Huang, Wanda Hung, and Kang G. Shin. FS^2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of 20th ACM Symposium on Operating System Principles*, October 2005.

[17] Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. *Computational Genomics*. Horizon, 2002. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology.

[18] Ozgur Can Leonard, Jason Neigh, Erez Zadok, Jefferey Osborn, Ariye Shater, and Charles Wright. The design and implementation of Elastic Quotas. Technical Report CUCS-014-02, Columbia University, June 2002.

[19] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, pages 275–288, 2002.

[20] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[21] James Mickens and Brian D. Noble. Exploiting availability prediction in distributed systems. In *Proceedings of the ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[22] Microsoft Corporation. `http://www.microsoft.com/technet/prodtechnol/winxppro/reskit/c28621675.mspx`.

[23] Ratul Mahajan Miguel. Controlling the cost of reliability in peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[24] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[25] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of USENIX Summer*, pages 247–256, 1990.

[26] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage*, 1(2):190–212, May 2005.

[27] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. Technical Report UCB/CSD-03-1299, EECS Department, University of California, Berkeley, 2003.

[28] Rodrigo Rodrigues and Barbara Liskov. High availability in DHTs: Erasure coding vs. replication. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems*, February 2005.

[29] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for

large-scale peer-to-peer systems. In *Proceedings of Middleware*, November 2001.

[30] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th SOSP (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[31] Margo Seltzer, Keith Bostic, Marshall Kirt McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Winter USENIX Technical Conference*, January 1993.

[32] Keith Smith and Margo Seltzer. File system aging. In *Proceedings of the 1997 Sigmetrics Conference*, Seattle, WA, June 1997.

[33] Daniel Stutzbach and Reza Rejaie. Towards a better understanding of churn in peer-to-peer networks. Technical Report UO-CIS-TR-04-06, Department of Computer Science, University of Oregon, November 2004.

[34] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: A mechanism for background transfers. *SIGOPS Oper. Syst. Rev.*, 36(SI):329–343, 2002.

Cobalt: Separating content distribution from authorization in distributed file systems

Kaushik Veeraraghavan, Andrew Myrick, and Jason Flinn
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

How should a distributed file system manage access to protected content? On one hand, distributed storage should make data access pervasive: authorized users should be able to access their data from any location. On the other hand, content protection is designed to *restrict* access — this is often accomplished by limiting the set of computers from which content can be accessed. In this paper, we propose a new method for storing content in distributed storage called Cobalt. Rather than grant access to data based on the computer that reads the data, Cobalt grants access based on the physical proximity of authorized users. Protected content is stored encrypted in the distributed Blue File System; files can only be decrypted through the cooperation of a personal, mobile device such as cell phone. The Cobalt device is verified by content providers: it acts as a proxy that protects their interests by only decrypting data when policies specified during content acquisition are satisfied. Wireless communication with the device is used to determine the physical proximity of its user; when the Cobalt device moves out of range, protected content is made inaccessible. Our results show that Cobalt adds only modest overhead to content acquisition and playback, yet it enables new forms of interaction such as the ability to access protected content on ad hoc media players and create playlists that adapt to the tastes of nearby users.

1 Introduction

The complexity of managing digital content continues to increase. Many people use a wide variety of computing and consumer electronics devices to access their content — PCs, laptops, MP3 players, and DVRs are just a few examples. While users typically access their content on a handful of well-known devices, inevitably some scenarios arise in which they would like to access their content on *ad hoc devices*, which we define to be devices that they do not own and do not commonly use. For instance, a visiting family member might wish to display photos using a living room DVR, or a party-goer might wish to share their taste in music by playing MP3 files on a friend's stereo.

Ad hoc access to digital content is challenging for several reasons. First, the ad hoc media player must locate the content that a user wishes to access. Currently, this requires the user to copy their content to portable storage or specify a location in a distributed storage system from which the content can be read. Second, if the content is not compartmentalized into a specific subtree of the portable or distributed storage, the media player must search through many files to locate relevant media. Over a wide-area link, such searches can be extremely time-consuming. Third, configuring a media player to locate the content might be challenging since each new device presents a different user interface. Finally, users may have to enter a password to grant the media player access to their content — however, they have no assurance that any entered password will not be abused.

Digital rights management introduces another dimension of complexity. Content providers who wish to ensure that users will not share their products in an unauthorized manner typically use some form of digital rights management. While many providers such as Yahoo [30] and the iTunes Music Store [1] allow a user to view content on multiple media players, the user must explicitly authorize the device on which content is viewed. In order to play protected content, the user must enter his userid and password. To revoke access to his content, the user must later deauthorize the ad hoc device. Potentially, such authorizations compromise privacy by informing the content provider of the movements and activities of users who have purchased their content.

In this paper, we introduce Cobalt, a mobile solution for ad hoc content access. Cobalt is implemented as an extension to the distributed Blue File System (BlueFS) [22]. Cobalt runs on a *token*, which is defined to be a mobile device such as a cell phone or PDA that is nearly always carried by its user. Cobalt improves usability and security by automating:

- **content location.** A Cobalt token automatically discovers media players in its immediate vicinity. When its user decides to access her content on an ad hoc player, she specifies her intentions as a semantic query. The token translates the semantic specification into a specific list of files to share. For example, a user may share all her MP3 files with a media player or just her recent vacation photos. The token provides the media player with the network address of the BlueFS server from which these files can be fetched, as well as a list of shared content.
- **authorization.** Cobalt uses techniques from Zero-Interaction Authentication [5] to limit the content that is shared with each media player. Content is protected with per-file keys that are encrypted with a key-encrypting key (KEK) known only to the token. Thus, the media player must contact the token to obtain the per-file key for all shared content. Unless the user has explicitly authorized sharing of the content with the media player, the Cobalt token denies access by refusing to decrypt the per-file key. Cobalt leverages Trusted Platform Module (TPM) hardware to ensure that decrypted per-file keys are not leaked by any media player to which they are provided. Currently, Cobalt limits ad hoc devices to read-only access to shared content.
- **digital rights management.** The token acts as a proxy for digital rights management that protects the interests of a content provider. During content acquisition, a provider uses the TPM to verify the integrity of the software running on the token. It then sends the token the key used to encrypt content along with a policy describing how that content can be played. The token encrypts the content key and a secure hash of the policy with its KEK. It stores these encrypted values along with the policy and encrypted content in BlueFS. Subsequently, it only decrypts the content key for media players that satisfy the policy. The use of a proxy improves usability because it eliminates the need to register and deregister media players with a content provider.

Our experimental results show that Cobalt adds only minimal overhead to content acquisition and playback. Further, this overhead does not substantially depend on the size of data being acquired or played. We also present results from a case study that shows how Cobalt can be used to enable new applications such as adaptive playlists that adjust the selection of music being played to match the tastes of people located nearby.

2 Design goals

In this section, we outline the goals that we followed in the design of Cobalt.

2.1 Usability

The primary design goal for Cobalt is usability. Cobalt minimizes the amount of effort required to access content. When user input is required, Cobalt strives to provide the interface on the token, with which the user is familiar, rather than on an ad hoc media player, which may have an unfamiliar interface.

Consider the effort that a typical user must currently exert to access protected content on an ad hoc media player. The user must authorize the new device with each content provider. Then, the user must provide the content to the device either with portable storage or by specifying a network address of a Web or distributed storage server from which data can be read. If the content is copied over the network and is not publicly accessible, the user must specify a password with which the media player can access the content. The user may need to manually create a playlist in order to specify what content should be shared with the ad hoc media player. Many of these interactions must be done using the unfamiliar media player interface.

For example, while iTunes allows content to be streamed to ad hoc computers, protected content can only be accessed after the userid and password of the person who owns the streaming computer is provided. Streaming is only permitted between computers on the same local network. While proxies exist that allow unpermitted access, setting up such proxies requires substantial configuration.

In contrast, Cobalt minimizes the actions required to share content with an ad hoc media player. It leverages a distributed file system, BlueFS, to share content seamlessly with the media player — this automatically provides prefetching and caching of content to improve the quality of playback. Since each token is associated with a specific BlueFS server, the user need not enter network addresses and other technical information. A Cobalt token allows its user to specify content to be shared as a semantic query rather than an explicit list of files; query results are automatically updated when a user adds content or modifies existing files. Finally, the token protects a user's content without the need to enter a password by decrypting only files that match the specified query.

2.2 Protection for content providers

The second design goal for Cobalt is to protect the interests of content providers. Content should not be leaked to unauthorized users or devices. Files should only be accessible on media players that satisfy the policy specified when content was acquired.

Figure 1 shows the Cobalt trust model. The token and ad hoc media player have Trusted Platform Modules. The

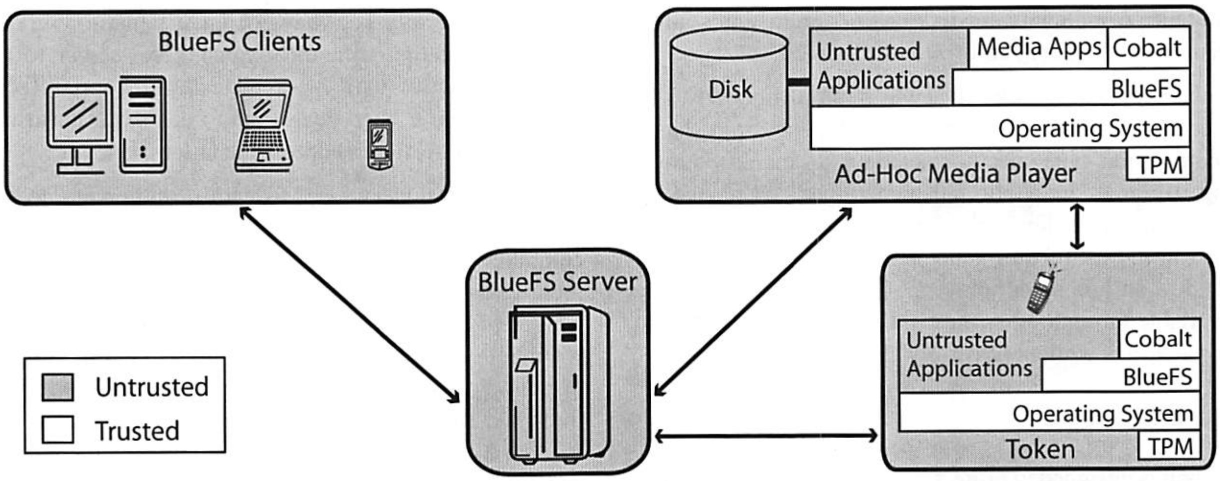


Figure 1. Cobalt trust model

TPM on each platform is used to verify the operating system, BlueFS client, and applications involved in content acquisition and playback. The TPM allows a content provider to verify the software running on the token — this verification is vital because the token acts as a proxy that protects the interests of the provider.

Using the media player's TPM, the token verifies the integrity of software on the player before it allows content to be decrypted. BlueFS, with the cooperation of the operating system, ensures that the decrypted content is only provided to the trusted application and not to other applications running on the media player. The operating system caches decrypted content in the buffer cache to improve performance. The BlueFS daemon may also cache content on disk; however, protected content on disk is always encrypted.

In contrast, the BlueFS server and its other clients are not trusted devices and do not need TPM hardware. The token has a symmetric key-encrypting-key (KEK) that is never exposed externally. Protected content is stored encrypted in BlueFS and cannot be decrypted without the KEK that is known only to the Cobalt token. This architecture allows unverified clients to prefetch and cache content on their local storage to improve performance. The Cobalt token only decrypts the content key for prefetched files after it has verified the trustworthiness of the media player.

Cobalt also allows users to protect their own content. For instance, a user might store all of her photos in BlueFS and protect them using Cobalt. When she visits a friend's house, she can then make specific subsets of those photos, e.g., vacation pictures, available on her friend's television. Photos that she does not choose to share will not be decrypted by her token.

A substantial advantage of this trust model is that the platforms required to have a TPM are typically closed. For example, the token may be a cell phone, and the media player may be a DVR or car stereo. In contrast to a general-purpose computer, these closed-platform devices typically run a much smaller set of software that may be more tightly controlled by their manufacturers. The task of verifying these platforms becomes easier given that the set of possible software and hardware combinations is limited. In contrast, the number of software and hardware combinations on a general-purpose computer is much larger, meaning that it might be more difficult to verify the integrity of other entities such as the BlueFS server.

2.3 Privacy

A Cobalt token preserves its user's privacy. The use of a mobile wireless device naturally raises concerns because the presence of such a device can be used to track the movements and activities of its user.

A Cobalt token discovers new media players in its vicinity without exposing its identity. The discovery protocol reveals to a media player that a token is located nearby; however, the discovery request is generic and reveals no information that can be used to identify a particular token. The user explicitly authorizes an interaction with a particular media player by selecting it from a menu on the token. Alternatively, the user may authorize future interactions with a media player by adding it to a list of pre-approved players. The token does not reveal its identity to unauthorized media players.

In contrast, if a user must register and deregister media players, the content provider has considerable information about the movement and activities of that person. In addition, if content is protected, then the user must disclose a password to the media player before accessing content. A malicious media player could use this information to access unauthorized information.

Cobalt currently has a privacy limitation. In order to let media players efficiently generate playlists from shared content, the type-specific metadata of shared content, e.g., ID3 tags, are public and unencrypted. We plan to address this limitation in the future by including the metadata in the list of files generated by the token and specified to ad hoc media players. Once such metadata are available through other means, they can be encrypted in content files without adversely affecting performance.

3 Threat model

Cobalt is designed to protect the interests of both its users and content providers. For individual users, Cobalt prevents unauthorized access to their data. An attacker may try to subvert Cobalt to obtain access to content that the user has not decided to share. For content providers, Cobalt prevents access to content in violation of the specified policy. An attacker may try to subvert Cobalt to gain access to content in a manner that violates the policy or attempt to make a copy of decrypted content to gain unfettered future access.

We assume that attackers are capable of monitoring all communication between parties such as the Cobalt token, content provider, media players, and BlueFS server. Cobalt uses the station-to-station protocol [6] to establish a session key to encrypt communication; we assume that the private keys used in this protocol are known only to the respective parties and that trusted mechanisms exist for obtaining the public keys of other parties. We further assume that the encryption used by Cobalt is strong enough to provide confidentiality and authentication.

We assume that an attacker cannot compromise the hardware on the Cobalt token or media players. An attacker who subverts these mechanisms can gain unfettered access to content. Similarly, we assume that an attacker cannot subvert software that has been certified as trusted. A software exploit in a token or media player could record content keys to give the attacker access to protected content. Keys might also be discovered through covert channels such as power analysis. Finally, the current TPM standard is known to be vulnerable if attackers can modify software after it has been loaded and verified by the TPM but before it is used to access content.

We assume that an attacker may gain access to data stored on disk. Content and keys written to persistent storage are always encrypted. On the token, we assume that the TPM's sealed storage mechanisms prevent an attacker from obtaining the encryption keys. An attacker who compromises the BlueFS file server does not gain access to content since all content is encrypted. However, the attacker may mount a denial-of-service attack by deleting content or causing the server to respond to queries with incomplete information. Techniques that have been developed to deal with untrusted file servers [14] might address the latter problem.

An attacker might attempt to gain unauthorized access to content through a wormhole attack [11] in which a computer near a media player forwards packets to a remote Cobalt token over the Internet and returns the token's replies to the media player. Cobalt defends against wormhole attacks with a protocol that requires tokens to periodically respond to challenge messages within a threshold time period. If a number of challenges are missed, a media player refuses to play content. We assume that a threshold value exists that allows nearby tokens to respond in time and that is also small enough to prevent responses to be received from remote tokens.

An attacker may gain the ability to play content by obtaining possession of a token. This does not give the attacker more privilege than the Cobalt user, so an attacker possessing a stolen token cannot make unauthorized copies of content. If a token is lost or stolen, a user may deauthorize the token by re-keying content stored in BlueFS. However, any content previously fetched and cached by an attacker would still be viewable. Some cell phones have locking mechanisms that require a user to enter a PIN or password before using the device — such mechanisms, while not required by Cobalt, could potentially reduce the damage caused by a lost token.

4 Background

Cobalt leverages prior work in distributed file systems and trusted computing. In order to put Cobalt in its proper context, we briefly describe the relevant details of this prior research.

4.1 Blue File System

BlueFS is a server-based distributed file system that is designed to meet the storage needs of small groups of individuals such as a family [23]. The BlueFS file server, which is assumed to have a static IP address, might reside in the family's home or with its ISP. BlueFS clients include traditional computers such as desktops and laptops, as well as consumer electronics appliances such as MP3 players, cell phones, DVRs, and digital cameras. BlueFS supports disconnected operation [12] for mobile clients. When clients are connected to the server, the BlueFS consistency model is similar to Coda's weakly connected mode [21]. Prior to this work, BlueFS clients were tightly bound to a single server; they could only read or write data stored by that server. In Section 5.4.2, we describe how we have extended BlueFS to support read-only sharing of data stored on different servers.

Cobalt exploits a novel feature of BlueFS called *persistent queries*. As described previously [23], a persistent query notifies standalone applications about modifications to data stored in the distributed file system. An application running on any client that is interested in receiving such notifications specifies a semantic query (e.g., all files that end in ".mp3") and the set of events in which it is interested (e.g., file existence and new file creation). The query is created as a new object within the file system. The BlueFS server evaluates the query and adds log records for matching events. For instance, in the above example, the server would initially add a log record to the query for every MP3 file accessible to the user who created the query and then incrementally add a new record every time a new MP3 file is created. Since the query and its results are a file system object, the underlying cache consistency mechanisms of BlueFS notify the application about changes to the query. Prior results have shown that persistent queries are fast to create since they are evaluated at the server, which keeps a metadata database to speed processing.

4.2 Trusted computing

Rather than propose a new model for trusted computing, Cobalt leverages previous work in this area [9, 15, 17]. It assumes that both the token and the ad hoc media player have a Trusted Platform Module, as defined by the Trusted Computing Group (TCG) [28]. In this section, we summarize only the portions of TPM that are relevant to our work: McCune et al. [17] provide a good introduction to TPM for those who wish more details.

A TPM implementation includes hardware support for cryptography primitives. At a minimum, Cobalt assumes that the TPM module incorporates two unique keys: the Attestation Identity Key (AIK) which is an RSA signing key pair and a symmetric Key-Encryption-Key (KEK). Any value signed with the AIK can be verified by an external entity using the public RSA key. The public RSA key may be exported as a certificate with an embedded chain that can be followed back to the manufacturer of the hardware. Thus, signing a value with the AIK allows any external entity to verify the identity and manufacturer of the device that generated the signature. To avoid computationally-expensive public key cryptography, the symmetric KEK is used to encrypt large data items.

Cobalt relies on TPM support for attestation. When requested, the TPM generates a *manifest* that explicitly lists the software loaded on the system, as well as a *Platform Configuration Registers (PCR) quote*, which is an AIK-signed hash of the manifest that can be used for verification. Upon request, a device provides its TPM-generated manifest and PCR quote to an external entity. The external entity verifies the manifest using the PCR quote and confirms that the software running on that device meets its approval. Additionally, if the entity doubts the integrity of the TPM on the device, it can verify the AIK signature on the PCR quote and trace the certificate chain to ensure that the device was manufactured by an entity that it trusts.

In Cobalt, content providers use the TPM to verify the integrity of a token. The token, in turn, uses the TPM to verify that media players meet the approval of policies specified by the provider during content acquisition.

The Cobalt token uses the KEK to encrypt the content key and hash of the policy that are given to it during content acquisition. Since the KEK never leaves the token, content keys cannot be subsequently decrypted without the cooperation of the token. We use the KEK to encrypt these data items because the performance of symmetric key encryption is substantially better than that of public key encryption. Our experimental results using a cell phone as the token confirm that public key operations can require several seconds to complete, whereas symmetric key operations require only a few milliseconds.

5 Implementation

5.1 Overall model

The Cobalt token is a small, mobile device. The token should be powerful enough to run a BlueFS client, yet small enough to always be carried by its user. Additionally, there should be a strong association between a token and its user so that the presence of the token can be taken to mean that its user is present. Cell phones are ideal Cobalt tokens because they meet all of the above criteria. Consequently, we use a Motorola E680i phone [20] for our token implementation in this paper. We have also ported the token to other platforms, such as the HP iPAQ PDA.

We assume that ad hoc media players run a BlueFS client. Given that platforms such as the TiVo DVR run the Linux operating system and have APIs that allow them to be extended with novel applications, we do not think this requirement will be onerous in the future. Our prior work [23] has also investigated how general-purpose computers can extend the functionality of consumer electronics appliances when those platforms are too closed to run a BlueFS client. Similar techniques could be applied in Cobalt if necessary (although the TPM verification would need to be extended to include the general-purpose computer).

Cobalt functionality can logically be separated into two phases: content acquisition, which is described in the next section, and content playback, which is discussed in Section 5.3. During content acquisition, the Cobalt token coordinates the acquisition of new protected content from a provider. The content, encrypted with a per-file content key, is stored in BlueFS. The content key, in turn, is encrypted with the token's KEK and also stored in BlueFS. Optionally, the content provider may supply a playback policy that is stored in BlueFS.

During playback, a token verifies that a file requested by an ad hoc media player has been authorized for access by its user. It checks that the media player meets the specifications provided by the content provider in the playback policy. It only decrypts the content key (allowing the ad hoc media player to decrypt the content) if both checks pass.

5.2 Acquiring content

The Cobalt token manages the acquisition of content. We have implemented content acquisition as a library routine that takes as parameters the network address of the content provider and a unique identifier for the specific content being acquired. Currently, we use a menu-driven user interface. However, our library design would make it trivial to substitute a more user-friendly GUI for directing content acquisition.

After the user selects the provider and content to acquire, the token opens a network socket to the specified provider. The token and provider mutually confirm each other's identities using the station-to-station protocol. Specifically, Cobalt uses the Full-STS variant that includes an exchange of public-key certificates. Each party provides the other with a certificate chain that vouches for their respective public keys. While Cobalt does not assume that the token and content provider have prior knowledge of each other, it does assume that there is at least one certificate authority trusted by each party that can vouch for the other. As a by-product of the station-to-station protocol, the token and the provider establish a symmetric session key that is used to encrypt further communication during content acquisition.

The provider verifies that the token is running software that it trusts. This is necessary since the provider will give the token sufficient information to decrypt the content. The provider therefore needs to verify that the token will only release the content in accordance with the specific policy for that content. For example, the content provider needs to assure itself that the token will not leak the content to an unauthorized third party.

The token uses its TPM's AIK as its public key during the station-to-station protocol. This allows the content provider to ascertain that the token has a valid TPM. The token then uses its TPM to generate a manifest and PCR quote. The content provider verifies the signature on the PCR quote and makes sure that the quote is a valid hash of the software manifest. The provider then verifies that the manifest entries are known versions of programs that can be trusted not to leak content.

The provider next encrypts the requested content with a symmetric content key. To improve performance, save battery lifetime, or deal with closed software environments in which the BlueFS client code cannot be run, the token typically enlists the cooperation of another *helper* computer during content acquisition. The helper is a BlueFS client that is known to the token; for example, it might be the user's workstation or BlueFS server. The helper need not be trusted by the content provider since it is never provided with the key necessary to decrypt the content. When a helper is used during content acquisition, the provider sends the encrypted content to the helper, which stores the data as a new file in BlueFS. If a helper is unavailable, the provider sends the content to the token. The token writes the data to BlueFS itself.

The provider then sends the content key and the policy for playing the content to the token. The policy for playback can be thought of as a set of requirements that the provider wishes to enforce on any device that tries to access the content. For example, the provider might request that the device be deployed on a TPM platform whose PCR quote corresponds to one of several known and trusted software configurations — this helps ensure that the content is not played on a device that leaks the content in an unauthorized manner. Alternatively, the policy might allow the provider, the manufacturer, or other trusted entities to vouch for software running on media players by digitally signing a certificate which lists the software. The media player could present a copy of this certificate to the token, which would verify that the certificate is signed by an entity trusted by the policy.

The token next generates a *protector* for the content being acquired. The protector is a concatenation of the content key and a SHA-1 hash of the policy specified by the provider. The token encrypts these values using its KEK and stores them in the BlueFS metadata of the newly-created content file. Cobalt uses AES Output-Feedback Chaining [7] when generating the protector—this ensures that the content key cannot be decrypted successfully if the policy hash is tampered with by an external entity. The token writes the policy, which may be of arbitrary length, to a separate file in BlueFS. The token deletes the content key from its memory after generating the protector. Although all our current tokens, the Motorola E680i cell phone and HP iPAQ, are BlueFS clients, Cobalt supports closed-platform tokens that cannot run the client code by allowing the helper to store the protector and policy on their behalf.

If a Cobalt user is protecting his own content, the above process is simplified. The user specifies files to protect using his token — these files can be in a private directory within BlueFS. The token generates a content key for each file, encrypts the data with that key, and stores the encrypted data in a new file in a publicly-accessible BlueFS directory. The protector is generated and stored as described above.

5.3 Playing content

The Cobalt token runs a wireless discovery protocol to learn about media players in its immediate vicinity. This protocol can be run periodically or on-demand when the user starts a media sharing application on the token. Periodic discovery reduces user-perceived latency, but on-demand discovery saves battery energy on the token by only performing discovery when necessary. During discovery, the token sends a broadcast message over the local network (currently, we use 802.11b in ad hoc mode for discovery). All media players on the local area network that are willing to access content respond to the token. If the media player is a TPM platform, its response includes its public key (AIK), the manifest of software running on the media player, and the signed PCR quote necessary to verify the manifest.

The token presents the user with a list of all media players that responded. When the user selects one of these players, the token and the media player mutually authenticate and establish a secure communication channel via the station-to-station protocol. Cobalt allows media players to reject connections from unknown tokens if they wish; however, our design does require the media player to disclose its identity to such tokens.

The user next specifies which content he is willing to share with the media player. This content is stored as encrypted, publicly accessible files in BlueFS. Rather than requiring the user to specify a lengthy list of files, the Cobalt token allows its user to semantically specify which content should be shared as a persistent query. For example, he can create a persistent query that matches all MP3 files or refine the query to specify only music files from a certain artist. To share content, the user may browse through current outstanding persistent queries and select one that is most appropriate. Alternatively, the user may create a new query that matches the content that they currently want to share. In either case, the user specifies the query using the interface of his token, i.e., one that he knows and is comfortable with, rather than an unfamiliar interface presented by an ad hoc media player After authenticating the media player and establishing a secure session, the token sends the media player the IP address of the user's BlueFS server and the unique 96-bit identifier of the persistent query that specifies the content to be shared.

The media player next contacts the user's BlueFS server and fetches the persistent query object. We have implemented a federation mechanism, described in the next section, that allows BlueFS clients to mount third-party servers as read-only directories within their distributed namespaces. The persistent query contains the unique identifier of all files that match the associated semantic string. Rather than search the entire BlueFS namespace for relevant files, the media player can read the query results to determine the exact set of files that are being shared. Then, the media player can read the metadata associated with these files and add them to its list of available content. The media player may prefetch and store locally some of this content to improve performance. However, until the token provides the content key for a file, the media player cannot decrypt cached content.

When a Cobalt-protected file stored in BlueFS is accessed, the file metadata indicates that the content is encrypted. On the first access to the file, the BlueFS client on the media player contacts the token to obtain the content key. Over the secure session established previously, the BlueFS client sends the token the protector encrypted with the KEK that includes both the policy hash and the content key. The BlueFS client on the media player also sends the token the policy for the file it wishes to read. The token decrypts the protector, hashes the specified policy, and verifies that the computed hash matches the one stored in the protector.

The token next verifies that the media player and its current software environment is in accordance with the policy for the specified content. As described by McCune et al. [17], the token can independently compute the PCR quote from the manifest and confirm that its computed quote matches the value supplied by the media player. If the software environment specified by the PCR quote is in accordance with the policy dictated by the content provider, then the token sends the decrypted content key back to the media player over the secure session. The media player uses this key to decrypt and play the content. Having verified the media player platform, the token trusts it not to leak decrypted content to a third party.

To limit interactions with the token, the media player caches content keys while the token is located nearby. Once a session is established, the player sends a challenge to the token every 30 seconds (the period is configurable). A prompt response to the challenge informs the media player that the token is still located within wireless communication range. If a number of consecutive challenges (currently, one) are not met, the media player assumes that the token has left its vicinity and destroys any keys cached on its behalf. Any decrypted content is flushed from the kernel buffer cache on the media player. This stops playback of the content.

The media player may continue to cache encrypted content on its local storage. This behavior improves performance by eliminating the need to refetch data from the BlueFS server if the token returns. It also potentially enables prefetching strategies (that we have not yet implemented) where a user's content can be prefetched by a media player in anticipation of her arrival and stored encrypted on a local disk for future use.

5.4 File system changes

5.4.1 Support for encryption

In order to support Cobalt, we made several changes to BlueFS. First, we added support for token-encrypted content. The metadata of each file stored in BlueFS can optionally contain two new fields: the protector that stores the encrypted policy hash and content key, as well as the unique BlueFS identifier of the token that can decrypt the protector.

Cobalt metadata fields can be set by any application executing with a userid that has write permission for a content file. The token first creates the file and writes the encrypted content using normal file system calls, then uses an IPC to add the metadata. Encrypted content is publicly readable — this allows an ad hoc media player to fetch and cache the content. However, the media player cannot decrypt the content without the cooperation of the token, nor can it modify the content. As mentioned in Section 2.3, file-specific metadata such as ID3 tags is unencrypted in our current implementation.

The BlueFS daemon only decrypts content immediately before providing it to the kernel as the result of a file read. If an encrypted content key is specified for a file, BlueFS verifies that it has established a secure session for the associated token. If no such session exists, it returns an error. Otherwise, it asks the token to decrypt the content key as described in Section 5.3. The content key is then used to decrypt the content.

To improve performance, a BlueFS client maintains a cache of decrypted keys for each connected token. All keys associated with a token are flushed from the cache if the specified number of consecutive challenge responses are not received from the token. At that time, the daemon also makes an upcall into the kernel to instruct the kernel module to flush any decrypted content associated with the flushed keys. Since content cached on storage devices is encrypted, no action is taken to destroy or evict on-disk files (however, files may be evicted later due to capacity constraints).

5.4.2 Support for read-only federation

Ordinarily, a BlueFS client is tightly bound to its server. A client registers with its server and receives a unique identifier. When a client caches a file, the server maintains a callback, which is a promise to notify the client if the file changes. Callbacks are maintained even when a client is disconnected to eliminate the need for full disk scans on reconnection.

The binding between an ad hoc media player and a BlueFS server must necessarily be more loose. We envision that the client running on the media player will be associated with the server of its owner. However, it must be able to read content from other servers in order to play the content of different users. We enable this loose binding through read-only federation.

We added a `federate` IPC to the BlueFS client that takes as input the IP address or hostname of a BlueFS server. When this function is invoked, the client connects to the server and assigns it a temporary *volume identifier* (like AFS [10] and Coda [12], BlueFS reserves the high-order 32-bits of its file identifier for an administrative volume identifier). The client maintains a mapping between the actual volume identifier that is permanently chosen by the server with which it is federated and the temporary volume identifier that has been assigned. After reading data from the server, the client changes the actual volume identifier to the temporary one. It makes the reverse change when sending requests to the server.

The server maintains callbacks for its federated clients so that it can maintain cache consistency when files change. However, in contrast to permanent clients for which it maintains persistent callbacks, the server immediately drops all callbacks held on behalf of a federated client once that client disconnects. Consequently, a client evicts all files that it has cached from a federated server as soon as it disconnects. Users may trigger an explicit disconnection with a `defederate` IPC.

Currently, federated clients may not modify files. While this may not suffice for all applications, read-only federation fits the needs of ad hoc content access. Media players can read public but encrypted content from federated servers and access the content with the cooperation of a Cobalt token. File updates are rare for media files. For those rare updates, e.g., a change to a song rating, a Cobalt user can make the change directly using his token, which is a permanent client of his server.

5.4.3 Support for other distributed storage

Although Cobalt was developed as an extension to BlueFS, one could potentially modify other distributed storage systems to use Cobalt. Cobalt has less than 5000 lines of source code, most of which could be reused.

The most significant change that would be required is for clients that run on ad hoc media players to be made Cobalt-aware. They need to establish secure sessions with mobile tokens and use the TPM to attest to the integrity of software running on the player. Such clients should restrict distribution of content to only authorized software components. A distributed storage solution must be able to associate Cobalt metadata (the policy, protector, and token identifier) with each file. It should also support a search mechanism, similar to persistent queries, that allows users to specify content to share.

For example, Cobalt could potentially be used with data stored on a Web server. One would need to build a Cobalt-aware Web client and provide a mechanism for a token to specify files to share. Cobalt metadata could be embedded in HTTP headers.

6 Evaluation

Our evaluation answers the following questions:

- What is the overhead of using Cobalt during content acquisition?
- What is the overhead of using Cobalt during content playback?
- Can Cobalt enable new applications?

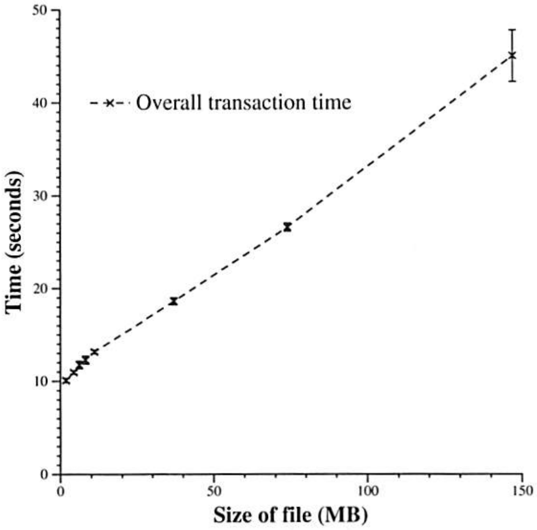
6.1 Methodology

In the following experiments, the Cobalt token is a Motorola E680i cell phone with a 300 MHz XScale processor. The cell phone is a BlueFS client that runs MontaVista Linux Consumer Electronics Edition 3.0. It communicates with the other computers via an SD/MMC 802.11b card. The BlueFS server runs on a Dell GX620 desktop with a 3.4 GHz Pentium 4 processor and 2 GB of DRAM. The desktop also runs a BlueFS client that is used during content acquisition. We use an IBM X40 laptop with a 1.2 GHz Pentium M processor and 784 MB of RAM as both the content provider and ad hoc media player. The laptop and desktop run Fedora Core 4 (Linux kernel 2.6.15) and are connected by 100 Mb/s Ethernet.

6.2 Content acquisition

We first measured the overhead that Cobalt adds to content acquisition. In this experiment, the token initiates the acquisition of a media file from the content provider. The BlueFS client running on the desktop is used as a helper during content acquisition, as described in Section 5.2.

Figure 2 shows how the total acquisition time varies with the size of the content being acquired. From the graph, it can be seen that there is an approximately 10 second fixed cost for acquiring a file plus a variable cost that is roughly linear with the size of the file. Table 1 shows detailed results for 2 MP3 audio files and 2 MP4 videos. For the smaller audio files, the majority of the acquisition time is used to establish a secure session between the token and the content provider. In these experiments, we assume that no prior relationship exists between these two parties; thus, each must send the other a certificate signed by a root authority to establish its identity.



This graph shows the time to acquire content for audio and video files of various sizes. Each result is the mean of 5 trials — the error bars are 90% confidence intervals.

Figure 2. Time to acquire content using Cobalt

The second column of Table 2 provides further detail about session establishment by detailing the time for the token to perform the individual components of the station-to-station protocol. This step is especially time-consuming since it requires the limited processor of the cell phone to perform public-key cryptography. For reference, Table 2 shows that if we replace the cell phone with the X40 laptop, session establishment requires an order of magnitude less time. Thus, as cell phone processors continue to improve, this component of the acquisition cost will diminish.

Once a secure session has been established, a symmetric session key is used for all further communication. Therefore, the session establishment time does not increase with the size of the file being transferred. Further, if multiple files are being acquired, the session need only be established once.

In contrast, Table 1 shows that the time to encrypt the content, transfer it to the helper, and store the data in BlueFS is roughly proportional to the size of the file being stored. For the larger video files, these activities comprise the majority of the time spent acquiring the content.

Operation	1.8 MB MP3	11 MB MP3	37 MB Video	147 MB Video
Secure session established	7.6 (±0.2)	7.5 (±0.3)	7.2 (±0.2)	7.5 (±0.3)
Content encrypted by provider	0.3 (±0.0)	1.8 (±0.0)	4.6 (±0.2)	14.3 (±0.1)
Content fetched and stored by helper	1.3 (±0.0)	3.0 (±0.2)	5.8 (±0.4)	22.3 (±2.6)
Metadata stored by token	0.9 (±0.1)	0.9 (±0.0)	1.0 (±0.2)	0.8 (±0.1)
Total acquisition time	10.1 (±0.2)	13.2 (±0.2)	18.6 (±0.4)	45.0(±2.8)

This table shows the time (in seconds) to acquire content using Cobalt for files of varying sizes. Each result is the mean of 5 trials — 90% confidence intervals are given in parentheses. The first row must be performed once per session, while the remaining rows are per-file costs.

Table 1. Time to acquire content using Cobalt

Operation	Cell phone (seconds)	Laptop (seconds)
Diffie-Hellman parameter generation	2.49 (±0.11)	0.14 (±0.00)
Preparation of signed exponentials and certificate encryption	3.17 (±0.17)	0.21 (±0.01)
Verification of certificate and signed exponentials	1.29 (±0.01)	0.06 (±0.00)
Other (exponential exchanges, key derivation, network, etc.)	0.60 (±0.01)	0.11 (±0.01)
Total session establishment time	7.56 (±0.19)	0.51 (±0.01)

This table shows the time to establish a secure session using the station-to-station protocol between the provider (Dell GX620) and both, the token (Motorola E680i) and the helper (IBM X40 laptop). Each value is the mean of five trials — 90% confidence intervals are given in parentheses.

Table 2. Detailed breakdown of the time to establish a secure session

However, these activities are not Cobalt-specific, as most existing methods for acquiring protected content must encrypt data, transmit it over the network, and store it on a destination computer. In fact, this experiment underestimates the network cost of content acquisition since the BlueFS desktop and the content provider communicate via local Ethernet. For instance, if the server's connection to the network were a 5 Mb/s cable link, transmitting the 147 MB file would take approximately four minutes. Thus, as network speeds decrease, Cobalt overhead becomes a smaller proportion of the total acquisition time.

The final activity, metadata creation by the token, consists of generating the protector and storing it in BlueFS. Since the metadata size is independent of the content size, the time to perform this activity is constant.

Overall, we are encouraged by these results since the overhead added by Cobalt (establishing the secure session and storing metadata) is less than 9 seconds and does not increase significantly with the size of the content being acquired. The vast majority of Cobalt overhead results from the need to establish a secure session between the content provider and token — this overhead will decrease as cell phone processors become more powerful.

6.3 Content playback

We next evaluated the time to access content using Cobalt. In this experiment, we use the X40 laptop to represent an ad hoc media player. The laptop runs a BlueFS client. It uses xmms to play MP3s and VLC to play video. To specify which files are shared with the media player, we created a persistent query that matched 1500 MP3 files stored on the BlueFS server.

Table 3 shows the time for a token to associate with a media player and specify the content that will be shared. The majority of the time is required to establish a secure session between the token and player. The secure session is necessary to confirm the identity of each party, since a media player may only wish to accept content from known sources, and the token must verify that the media player is a trusted platform that meets the security policy for the content being shared. As part of session establishment, the token receives and caches the media player's PCR quote and software manifest, as provided by the player's TPM hardware. Since our laptop does not have TPM hardware, the laptop transmits precomputed values (a SHA-1 hash for the PCR quote and 1 KB file for the manifest) to the token on request.

Cobalt takes 4 seconds to create a persistent query specifying the content to be shared — most of this time is spent resolving the path names associated with the media files in the query. Currently, each path resolution requires multiple remote procedure calls to the server — based on these results, we are currently considering methods for pipelining these operations to reduce latency.

Table 4 shows the per-file costs for playing content once the token has associated with an ad hoc media player. When the first 4 KB data block is read from a Cobalt-protected file, the BlueFS client running on the media player asks the token to decrypt the content key. Thus,

Operation	Time (seconds)
Secure session establishment	7.9 (±0.2)
Media player selection and TPM verification	0.2 (±0.0)
Persistent query creation and content path resolution	4.0 (±0.2)
Playlist creation	0.3 (±0.1)
Total association time	12.3 (±0.3)

This figure shows how long it takes a Cobalt token to associate with a nearby media player and create a playlist with 1500 MP3s. Each result is the mean of five trials — 90% confidence intervals are given in parentheses.

Table 3. Time for the token to associate with a media player

Operation	Time (seconds)
First block decryption time	0.273 (±0.013)
Subsequent block decryption time	0.001 (±0.000)

This table shows the time to decrypt content when playing it on an ad hoc media player. The first 4 KB block decryption time includes the time for the token to verify the policy and decrypt the content key. Decryption of subsequent 4 KB blocks is much quicker since the media player caches decrypted content keys. Each result is the mean of five trials — 90% confidence intervals are given in parentheses.

Table 4. Decryption time

the first block decryption time includes the time taken by the token to decrypt the protector, verify the policy hash, check the policy against the media player's PCR quote and manifest, and return the content key if all checks pass. Cobalt decrypts the first file block in 273 ms, while it takes only 1 ms to decrypt subsequent blocks since the content key is cached.

We have experimentally verified that Cobalt successfully prevents the media player from decrypting content stored on the federated BlueFS server that is not explicitly shared by the persistent query (e.g., in the above experiment, the media player is unable to play the video files since only music files were shared). We have also verified that the token prevents the media player from decrypting content when its manifest does meet the policy specified for a given file. Finally, our results show that when the token leaves the vicinity of the media player, playback of the video ceases after approximately 30 seconds due to the absence of challenge responses.

6.4 Decryption CPU load

We used the `top` utility to measure the CPU consumption of Cobalt while decrypted content is accessed. During playback of the video, Cobalt consumes 2.1% of the CPU on the laptop while the VLC application uses 10.9% of the CPU. When MP3 audio files are played, Cobalt consumes 0.7% of the CPU while xmms uses 0.3% of the CPU. These results match our expectations: since video files have a higher data rate than music files, Cobalt must decrypt more data per second for the videos.

6.5 Case study: Adaptive playlists

We also explored a potential new application that Cobalt enables. Typically, when visiting a friend's house, the only content available is that which resides in the friend's music collection. However, with Cobalt, guests can pool their content to create a more diverse set of music. With this greater pool of potential content comes a problem: not all music may be enjoyable to everyone in the room.

To address this scenario, we built a Cobalt application that uses persistent queries to play only content that is mutually enjoyable to all people located nearby. Each user's Cobalt token sends a persistent query that lists their most highly rated songs to the media player. The media player compares the results of all queries and creates an adaptive playlist that consists only of songs that are in a specified number of query results (currently, all of them). This application assumes that there is uniformity in labeling MP3s, which seems reasonable given the availability of ID3 repositories such as freedb [8].

Table 5 shows the time for the media player to create an adaptive playlist. In this experiment, the owner of the media player specifies a query that matches on 650 songs. The owner of the Cobalt token specifies a query that matches on 1500 songs. When these queries are combined, the adaptive playlist consists of 650 songs that were included in both query results. Comparing the results in Table 5 with those in Table 3, it is apparent that creating the adaptive playlist takes only about a second longer than creating one based solely on remote content. The extra time is required to create a persistent query for the media player's owner.

Operation	Time (seconds)
Secure session establishment	7.7 (±0.2)
Media Player selection and TPM verification	0.3 (±0.0)
Local persistent query creation and path resolution	1.0 (±0.0)
Remote persistent query evaluation and path resolution	4.1 (±0.0)
Merged playlist creation	0.2 (±0.0)
Total time to create a new adaptive playlist	13.2 (±0.2)

This table shows the time needed by a guest's token and an ad hoc media player to create a new adaptive playlist with 650 MP3s from a collection of 1500 MP3s. Each value is the mean of 5 trials — 90% confidence intervals are given in parentheses.

Table 5. Time to create an adaptive playlist

7 Related work

To the best of our knowledge, Cobalt is the first system to use a mobile token to assist in the secure playback of protected content on ad hoc media players. Specifically, Cobalt separates content distribution from authorization by utilizing a distributed file system (BlueFS) as the distribution channel and a mobile device such as a cell phone to perform authorization on behalf of a content provider.

Content secured by Cobalt is protected by a digital container as described by Sibert et al. [25]. Cobalt extends this scheme by storing the content key in a trusted mobile device that can be used to decrypt the content when requested by the user.

Cobalt builds on Zero-Interaction Authentication [5]. ZIA introduced the notion of proximity-based encryption in which users carry a wearable token that announces their presence to their mobile computer. The token exchanges periodic messages with the computer to confirm its presence. If a user moves away from her computer, ZIA encrypts sensitive data stored on disk and in memory. When the user returns, ZIA decrypts the data so that it can again be accessed.

Cobalt differs from ZIA by focusing on scenarios where a single user does not own all computers. Cobalt uses TPMs to let content providers verify the integrity of a token and to allow the token to validate the integrity of a media player. Further, Cobalt tokens can dynamically associate with ad hoc media players. In contrast to ZIA which decrypts all files in a user's presence, Cobalt users can scope the specific files that they wish to decrypt using persistent queries.

More generally, Cobalt is an example of *splitting trust* [3, 26], in which a small, trusted device performs certain critical functions while a more resourceful computer executes the more demanding part of an application.

Pierce and Mahaney [24] have advocated using cell phones to perform additional functionality for usability reasons; Cobalt follows their advice in that it allows its user to interact with the system via their phone rather than through the interface of an ad hoc media player.

Cobalt assumes that tokens and ad hoc media players are deployed on a trusted computing platform that meets the Trusted Platform Module standard [28] defined by the Trusted Computing Group [27]. Currently, this standard is being extended to better support mobile devices such as the Cobalt token [19]. BitE [17] extends the TPM architecture by showing how a software manifest and PCR quote can be used to verify the integrity of a trusted device. Cobalt leverages these ideas when it checks the integrity of the token and media player. BitE provides a secure channel through which a user can enter sensitive data using their phone. Since Cobalt focuses on integrating protected content and distributed storage, the manner in which it uses the manifest and PCR quote is different from how they are used in BitE. We have tried to make Cobalt as agnostic as possible with regard to the trusted platform on which it runs; potentially, this could enable Cobalt to run on alternative architectures such as XOM [15] and Terra [9].

Commercial systems such as Apple's iTunes Music Store, Yahoo's Music Unlimited and Microsoft's Zune also deploy content protection mechanisms. Subscription-based services, such as Yahoo! Music Unlimited [30], allow users to play content on ad hoc media players after explicitly authenticating with a userid and password. Apple's iTunes Music Store [1] allows a user to purchase protected content and stream it to an ad hoc media player. Playback commences only after the user provides her Apple userid and password to the iTunes application on the ad hoc media player [2]. In contrast, Cobalt improves usability as it removes the requirement of entering a userid and password. Additionally, Cobalt improves privacy as the content provider is not informed every time the user accesses content from an ad hoc media player. Finally, Cobalt better protects the interests of content providers as the physical proximity of a Cobalt token such as a cell phone is a better indicator of the presence of the content owner than a password, which can be entered by another person.

Microsoft's Zune can wirelessly discover another Zune in its vicinity and share media. However, Zune places usage restrictions such as disallowing repeated reception of the same content and limiting the number of times shared content is played [18]. In comparison to Cobalt, Zune operates on a different model where content acquisition, not playback, is proximity-based. Unlike Zune whose playback policy is based on the number of accesses, Cobalt permits playback of protected content from ad hoc media players as long as the Cobalt token is in its vicinity.

Many distributed file systems support a form of federation. For instance, the Self-certifying File System [16] supports secure federation through symbolic links that include the public key of the federated server. AFS [10] has long supported a global namespace. Coda [12] has been recently enhanced to enable clients to access data in more than one cell, and the Glamour project [29] has added federation to NFSv4. Other file systems such as NFSv3 [4] and CIFS [13] allow ad hoc clients to connect to arbitrary servers. Cobalt could potentially leverage these federation mechanisms to allow ad hoc media players to access data from other file systems.

8 Future work

In the future, we hope to investigate what other novel applications are enabled by Cobalt. The presence of personal mobile devices such as cell phones provides valuable context about which people are physically present. These devices can inform their environment about the tastes and preferences of their users, allowing customization of pervasive applications. Adaptive playlists are one example of such customization. Alternatively, the Cobalt token could record which content has been recently played in the presence of its user to avoid repeats or help start playback of content such as movies at the place where its user last left off. Context could be used to manage deletion of old content; for instance, a DVR might delay automatic deletion of a TV show until all users who typically watch that show have viewed it.

We also would like to explore prefetching and caching policies for ad hoc media players. Given sufficient storage, a media player may choose to hide network delays by fetching encrypted content before it is played (perhaps using a playlist to anticipate what might be played). The media player may also choose to cache encrypted content in case a visitor who has recently departed returns in the future. While ad hoc media players are currently limited to read-only access by the BlueFS federation mechanism, we are considering adding the ability for a token to grant update permission by signing and passing a capability for the authorized access to a media player.

Cobalt currently supports only one token per file. We plan to support more than one token for a file by storing multiple token identifiers and protectors in the BlueFS metadata. Use of Cobalt does not preclude a user from authorizing additional devices that they own to play content using existing provider mechanisms. For instance, a user might authorize a home computer and download content directly to its hard drive, then also store a copy of the content using Cobalt so that it can be accessed on ad hoc media players.

Finally, Cobalt currently assumes a one-size-fits-all policy for how content is invalidated when the token is inaccessible. A better solution would be to have the per-file policy specify the amount of time that a file would remain valid after the token departs. Potentially, the per-file policy could also specify the minimum frequency for challenge-response messages and the maximum number of consecutive responses that can be missed before the token is assumed to no longer be present.

9 Conclusion

The goals of content protection often conflict with those of a distributed file system: the former is designed to make data less accessible, while the latter is designed to make data more accessible. Cobalt is targeted at reaching a reasonable compromise between these two goals that meets the needs of both users and content providers. Cobalt bases its authorization on the physical presence of a user and leverages a personal mobile device such as a cell phone to determine when a user is located nearby. Our results show that the overhead of Cobalt is quite reasonable. Our case study shows that Cobalt can also add value by enabling context-sensitive applications such as adaptive playlists.

Acknowledgments

We thank Ed Nightingale and Dan Peek for their help with BlueFS. Manish Anand, Evan Cooke, our shepherd, Mike Swift, and the anonymous reviewers provided valuable feedback about these ideas. The work is supported by the National Science Foundation under award CNS-0306251. Jason Flinn is supported by NSF CAREER award CNS-0346686. Intel Corp. and Motorola Corp. have provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, Motorola, the University of Michigan, or the U.S. government.

References

[1] APPLE. iTunes Music Store Customer Service - Authorizing your computer. http://www.apple.com/support/itunes/musicstore/authorization/.

[2] APPLE. iTunes Tutorial - Sharing your music on your local network. http://www.apple.com/support/itunes/windows/tutorial/segment102094b.html%1.

[3] BALFANZ, D., AND FELTEN, E. W. Hand-held computers can be better smart cards. In *Proceedings of the 1999 USENIX Security Symposium* (1999).

[4] CALLAGHAN, B., PAWLOWSKI, B., AND STAUBACH, P. NFS Version 3 Protocol Specification. Tech. Rep. RFC 1813, IETF, June 1995.

[5] CORNER, M. D., AND NOBLE, B. D. Zero-interaction authentication. In *Proceedings of the 8th International Conference on Mobile Computing and Networking* (September 2002), pp. 1–11.

[6] DIFFIE, W., VAN OORSCHOT, P. C., AND WIENER, M. J. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography 2*, 2 (1992), 107–125.

[7] DWORKIN, M. NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation. Tech. rep., National Institute of Standards and Technology (NIST), 2001.

[8] freedb. http://www.freedb.org.

[9] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 193–206.

[10] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[11] HU, Y.-C., PERRIG, A., AND JOHNSON, D. B. Wormhole attacks in wireless networks. *IEEE Journal on Selected Areas in Communications 24*, 2 (February 2006), 370–380.

[12] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[13] LEACH, P., AND PERRY, D. CIFS: A Common Internet File System. In *Microsoft Interactive Developer* (November 1996).

[14] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 121–136.

[15] LIE, D., THEKKATH, C. A., AND HOROWITZ, M. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, 2003), pp. 178–192.

[16] MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, December 1999), pp. 124–139.

[17] MCCUNE, J. M., PERRIG, A., AND REITER, M. K. Bump in the ether: A framework for securing sensitive user input. In *Proceedings of the USENIX 2006 Annual Technical Conference* (Boston, MA, June 2006).

[18] MICROSOFT. Share Audio Files Zune to Zune. http://www.zune.net/en-us/support/howto/zunetozune/sharesongs.htm.

[19] Mobile Device Security and Trusted Computing - Next Steps. Tech. rep., Trusted Computing Group, 2005. https://www.trustedcomputinggroup.org/groups/mobile.

[20] MOTOROLA. *Motorola E680*, September 2004. http://www.motorola.com/us/products.jsp.

[21] MUMMERT, L., EBLING, M., AND SATYANARAYANAN, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995).

[22] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 363–378.

[23] PEEK, D., AND FLINN, J. EnsemBlue: Integrating consumer electronics and distributed storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, November 2006), pp. 219–232.

[24] PIERCE, J. S., AND MAHANEY, H. Opportunistic annexing for handheld devices: Opportunities and challenges. In *Proceedings of HCIC* (2004).

[25] SIBERT, O., BERNSTEIN, D., AND WIE, D. V. Digibox: A self-protecting container for information commerce. In *Proceedings of the first USENIX Workshop on Electronic Commerce* (New York, New York, 1995).

[26] STAJANO, F., AND ANDERSON, R. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proceedings of the 7th International Workshop on Security Protocols* (1999), pp. 172–194.

[27] TCG PC Specific Implementation Specification v1.1. Tech. rep., Trusted Computing Group, 2006.

[28] TCG TPM Specification Version 1.2 Revision 94. Tech. rep., Trusted Computing Group, March 2006. https://www.trustedcomputinggroup.org/specs/TPM.

[29] TEWARI, R., HASWELL, J. M., NAIK, M. P., AND PARKES, S. M. Glamour: A Wide-Area Filesystem Middleware Using NFSv4. Tech. Rep. RJ10368, IBM, June 2005.

[30] Yahoo! Music Unlimited. http://music.yahoo.com/unlimited/.

PARAID: A Gear-Shifting Power-Aware RAID

Charles Weddle, Mathew Oldham, Jin Qian, An-I Andy Wang,
Florida State University,{weddle, oldham, qian, awang}@cs.fsu.edu
Peter Reiher, *University of California, Los Angeles,reiher@cs.ucla.edu*
Geoff Kuenning, *Harvey Mudd College, geoff@cs.hmc.edu*

Abstract

Reducing power consumption for server computers is important, since increased energy usage causes increased heat dissipation, greater cooling requirements, reduced computational density, and higher operating costs. For a typical data center, storage accounts for 27% of energy consumption. Conventional server-class RAIDs cannot easily reduce power because loads are balanced to use all disks even for light loads.

We have built the Power-Aware RAID (PARAID), which reduces energy use of commodity server-class disks without specialized hardware. PARAID uses a skewed striping pattern to adapt to the system load by varying the number of powered disks. By spinning disks down during light loads, PARAID can reduce power consumption, while still meeting performance demands, by matching the number of powered disks to the system load. Reliability is achieved by limiting disk power cycles and using different RAID encoding schemes. Based on our five-disk prototype, PARAID uses up to 34% less power than conventional RAIDs, while achieving similar performance and reliability.

1 Introduction

The disk remains a significant source of power usage in modern systems. In Web servers, disks typically account for 24% of the power usage; in proxy servers, 77% [CARR03, HUAN03]. Storage devices can account for as much as 27% of the electricity cost in a typical data center [ZHU04]. The energy spent to operate disks also has a cascading effect on other operating costs. Greater energy consumption leads to more heat dissipation, which in turn leads to greater cooling requirements [MOOR05]. The combined effect also limits the density of computer racks, which leads to more space requirements and thus higher operating costs.

Data centers that use large amounts of energy tend to rely on RAID to store much of their data, so improving the energy efficiency of RAID devices is a promising energy-reduction approach for such installations. Achieving power savings on commodity server-class disks is challenging for many reasons: (1) RAID performance and reliability must be retained for a solution to be an acceptable alternative. (2) To reduce power, a server cannot rely on caching and powering off disks during idle times because such opportunities are not as frequent on servers [GURU03, CARR03, ZHU04]. (3) Conventional RAID balances the load across all disks in the array for maximized disk parallelism and performance [PATT88], which means that all disks are spinning even under a light load. To reduce power consumption, we must create opportunities to power off individual disks. (4) Many legacy reliability encoding schemes rely on data and error-recovery blocks distributed among disks in constrained ways to avoid correlated failures. A solution needs to retrofit legacy reliability encoding schemes transparently. (5) Server-class disks are not designed for frequent power cycles, which reduce life expectancy significantly. Therefore, a solution needs to use a limited number of power cycles to achieve significant energy savings.

Some existing approaches use powered-down RAIDs for archives [COLA02] and trade performance for energy savings [PINH01]. Some studies have exploited special hardware such as multi-speed disks [CARR03, LI04, ZHU05]. Although simulation studies show promising energy savings, multi-speed disks are still far from ubiquitous in large-scale deployments [LI04, YAO06]. With the aid of nonvolatile RAM, approaches that use existing server-class drives have been recently made available [LI04, YAO06, PINH06], but the RAID reliability encoding constraints limit the number of spun-down drives (e.g. one for RAID-5).

We have designed, implemented, and measured the Power-Aware RAID (PARAID), which is deployable with commodity server-class disk drives, without special hardware. PARAID introduces a skewed striping pattern that allows RAID devices to use just enough disks to meet the system load. PARAID can vary the number of powered-on disks by *gear-shifting* or switching among sets of disks to reduce power consumption. Compared to a conventional 5-disk RAID, PARAID can reduce power consumption by up to 34%, while maintaining comparable performance and reliability. Moreover, PARAID reuses different RAID levels so that the underlying RAID technology can evolve independently.

Beyond the power savings obtained by PARAID, the process of creating a real energy measurement framework produced some useful insights into the general problem of measuring energy consumption and savings. These insights are also discussed in this paper.

2 Observations

Over-provisioned resources under RAID: Load balancing allows a conventional RAID device to maximize disk parallelism and performance, and ensures that no disk becomes a bottleneck. This uniformity simplifies data management and allows all disks to be accessed in the same way. However, uniform striping is not favorable for energy savings. Load balancing significantly reduces opportunities to power off disks because all disks in the array need to be powered to serve a file, even if a RAID receives relatively light loads, when fewer powered disks would be sufficient.

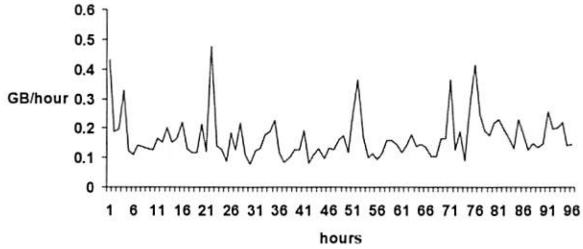


Figure 2.1: UCLA Computer Science Department web server activity from August 11 through August 14, 2006.

Cyclic fluctuating load: Many system loads display cyclic fluctuations [CHAS01]. Figure 2.1 shows the web traffic gathered at the UCLA Computer Science Department across one week. The load fluctuations roughly follow daily cycles. Depending on the types of traffic, different systems may exhibit different fluctuation patterns, with varying ranges of light to heavy loads [IYEN00].

We can exploit these patterns by varying the number of powered disks, while still meeting performance needs and minimizing the number of power switches. A few strategically timed power cycles can achieve significant power savings.

Unused storage space: Storage capacity is outgrowing demand for many computing environments, and various large-scale installations report only 30% to 60% storage allocation [ASAR05, GRAY05, LEVI06]. Researchers have been looking for creative ways to use the free storage (e.g. trading off capacity for performance [YU00] and storing every version of file updates [SANT99]).

Additionally, many companies purchase storage with performance as the top criterion. Therefore, they may need many disks for parallelism to aggregate bandwidth, while the associated space is left largely unused. Further, administrators tend to purchase more space in advance to avoid frequent upgrades. Unused storage can then be used opportunistically for data-block replication to help reduce power consumption.

Performance versus energy optimizations: Performance benefits are realized only when a system is under a heavy load, and may not result in an immediate monetary return. Energy savings, however, are available at once, and could, for example, be invested in more computers. Also, unlike performance, which is purchased in chunks as new machines are acquired, energy savings can be invested immediately and compounded over the lifetime of the computers. Therefore, if a server usually operates below its peak load, optimizing energy efficiency is attractive.

3 Power-Aware RAID

The main design issues for PARAID are how to skew disk striping to allow opportunities for energy savings and how to preserve performance and reliability.

3.1 Skewed Striping for Energy Savings

PARAID exploits unused storage to replicate and stripe data blocks in a skewed fashion, so that disks can be organized into hierarchical overlapping sets of RAIDs. Each set contains a different number of disks, and can serve all requests via either its data blocks or replicated blocks. Each set is analogous to a gear in automobiles, since different numbers of disks offer different levels of parallelism and aggregate disk bandwidth.

The replicated blocks are soft states, in the sense that they can be easily reproduced. Thus, as storage demands rise, replicated blocks can be reclaimed by reducing the number of gears. Unlike memory caches, these soft states persist across reboots.

Figure 3.1.1 shows an example of replicated data blocks persisting in soft states in the unused disk regions. By organizing disks into gears, PARAID can operate in different modes. When operating in gear 1, with disks 1 and 2 powered, disks 3 and 4 can be powered off. As the load increases, PARAID up-shifts into second gear by powering up the third disk.

By adjusting the number of gears and the number of disks in each gear, PARAID provisions disk parallelism and bandwidth so as to follow the fluctuating performance demand curve closely through the day. By creating opportunities to spin down disk drives, PARAID conserves power.

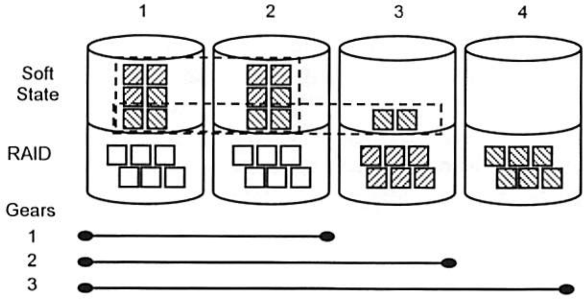


Figure 3.1.1: Skewed striping of replicated blocks in soft state, creating three RAID gears over four disks.

While more gears can match the performance demand curve more closely, the number of gears is constrained by the unused storage available and the need for update propagation when switching gears. To minimize overhead, the gear configuration also needs to consider the number of gears and gear switches.

3.2 Preserving Peak Performance

PARAID matches the peak performance of conventional RAIDs by preserving the original disk layouts when operating at the highest gear. This constraint also allows PARAID to introduce minimal disturbances to the data path when the highest gear is in use.

In low gears, since PARAID offers less parallelism, the bandwidth offered is less than that of a conventional RAID. Fortunately, the number of requests affected by this performance degradation is significantly smaller compared to those affected during peak hours. Also, as bandwidth demand increases, PARAID will up-shift the gear to increase disk parallelism.

PARAID also can potentially improve performance in low-gear settings. As a gear downshifts, the transfer of data to the soft state from disks about to be spun down warms up the cache, thus reducing the effect of seeking between blocks stored in different gears.

3.3 Retaining Reliability

To retain conventional RAID reliability, PARAID must be able to tolerate disk failures. To accomplish this goal, PARAID needs to supply the data redundancy of conventional RAIDs and address the reduced life expectancy of server-class disks due to power cycles.

PARAID is designed to be a device layer sitting between an arbitrary RAID device and its physical devices. Thus, PARAID inherits the level of data redundancy, striping granularity, and disk layout for the highest gear provided by that RAID. For example, a PARAID device composed with a RAID-5 device would still be able to rebuild a lost disk in the event of disk failure. (The details of failure recovery will be discussed in Section 4.4.)

Because PARAID power-cycles disks to save energy, it must also address a new reliability concern. Power-cycling reduces the MTTF of a disk, which is designed for an expected number of cycles during its lifetime. For example, the disks used in this work have a 20,000-power-cycle rating [FUJI05]. Every time a disk is power-cycled, it comes closer to eventual failure.

PARAID limits the power cycling of the disks by inducing a bimodal distribution of busy and idle disks. The busier disks stay powered on, and the more idle disks often stay off, leaving a set of middle-range disks that are power-cycled more frequently. PARAID can then prolong the MTTF of a PARAID device as a whole by rotating the gear-membership role of the disks and balancing their current number of power cycles.

Further, PARAID limits the power cycles for disks. By rationing power cycles, PARAID can operate with an eye to targeted life expectancy. For example, if the disks have a five-year life expectancy due to the system upgrade policy, and the disks are expected to tolerate 20,000 cycles, then each disk in the array cannot be power cycled more than 10 times a day. Once any of the disks has reached the rationed numbers of power cycles for a given period, PARAID can operate at the highest gear without energy savings. The power-saving mode resumes at the next rationing period.

4 PARAID Components

PARAID has four major components—a block handler, monitor, reliability manager, and disk manager (Figure 4.1)—responsible for handling block I/Os, replication, gear shifting, update propagation, and reliability.

4.1 Disk Layout and Data Flow

PARAID is a new device layer in the conventional software RAID multi-device driver. The block handler under PARAID transparently remaps requests from a conventional RAID device and forwards them to other soft-state RAID devices or individual disk devices.

PARAID currently delegates RAID regions to store replicated soft states for individual gears. The highest gear reuses the original RAID level and disk layout to preserve the peak performance. When the highest gear is active, PARAID forwards requests and replies with minimal disturbance to the data path.

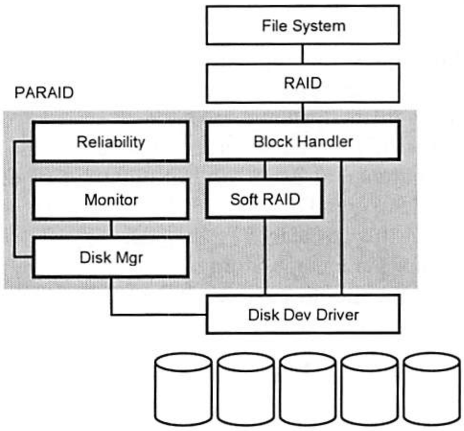


Figure 4.1: PARAID system components.

However, the data and parity blocks of D disks cannot be striped across fewer disks to achieve the same level of redundancy. If we simply assigned the Dth block to one of the still-powered disks, it would be possible for a single drive to lose both a data block and a parity block from the same stripe, while the block stored on the powered-off disk might be out of date.

To provide reliability, the soft-state replicated blocks stored in each gear use the same RAID level. For example, consider a 5-disk RAID-5 (Table 4.1). Gear 2 uses all 5 disks; gear 1 uses 4. When disk 5 is spun down, its blocks must be stored on the remaining 4 disks. This is done by creating a 4-disk soft-state RAID-5 partition; the data and parity blocks from disk 5 are stored in this partition as if they were normal data blocks arriving directly from the application. If necessary, the soft-state partition can be removed to recover space whenever disk 5 is spinning.

The synchronization between disk 5 of gear 2 and the blocks in gear 1 resembles the data flow of RAID1+0. Disk 5 is "mirrored" using RAID-5 on gear 1, with synchronization performed during gear shifts.

By using the underlying RAID-5 code for disk layout and parity handling, the PARAID code is drastically simplified compared to trying to deal with those details internally.

	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
Gear 1	(1-4)	8	12	((1-4),8,12)	
RAID-5	16	20	(16,20,_)	_	
	1	2	3	4	(1-4)
Gear 2	5	6	7	(5-8)	8
RAID-5	9	10	(9-12)	11	12
	13	(13-16)	14	15	16
	(17-20)	17	18	19	20

Table 4.1: PARAID disk layout with one 4-disk gear and one 5-disk gear, each running RAID-5. Each table entry contains either a block number or numbers enclosed with parentheses, denoting a parity block. "_" means an empty block.

For all gears (including the case where all disks are powered), if either a read or write request is sent to a powered disk, the disk simply serves the request. If a request is sent to a powered-off disk, then PARAID will remap the request to a replicated block stored on a powered disk. A remapped update is later propagated to neighboring gears during gear shifts.

The required unused storage depends on the RAID level, the number of gears, and the number of disks in each gear. For RAID-5, $D > 3$ disks, M gears with G_i disks within the ith gear ($1 \leq i \leq M$, $3 \leq G_i < G_{i+1} < G_M = D$) the percentage storage consumption S_i of the total RAID for the ith gear can be solved with M equations:

$$\begin{cases} \sum_{i=1}^{M} S_i = 1 \\ \left(\sum_{j=i}^{M} S_j\right)(G_i - G_{i-1}) = (G_{i-1} - 1)S_{i-1}, i = 2..M \end{cases} \quad (1)$$

For a disk in the lowest gear (Figure 3.1.1, disk 1), the sum of the percentage usage of disk space by each gear must be one. Also, for a gear (Figure 3.1.1, gear 2) to be able to shift to a lower gear (Figure 3.1.1, gear 1), the lower gear must store all the content of the disk(s) (Figure 3.1.1, disk 3) that are about to be spun down, with their parity information created for the lower gear.

PARAID uses around $(D - G_1)/(D - 1)$ of the total RAID-5 storage to store soft states. This estimate is largely based on the number of disks in the lowest gear, not the number of gears or the number of disks in intermediate gears, so the overhead of gear switching and the time spent in each gear will determine optimal gear configurations.

The target percentage of energy savings for an active system (not specific to RAID-5) is described by formula (2), where $P_{standby}$ is the power consumption for a spun-down disk (more details are given in the performance section), and $P_{active/idle}$ is the average power consumption for either a busy disk or an idle disk, to compute disk power savings for busy or idle loads.

Power savings increase with more disks, fewer disks in the lowest gear, and a higher $P_{active}/P_{standby}$ ratio. Since spun-down disks still consume power, it is better to install PARAID with large disks with unused space, rather than buying more disks later.

$$1 - \frac{(D - G_1)P_{standby} + G_1 P_{active/idle}}{DP_{active/idle}} \quad (2)$$

For this paper, an up-shift means switching from a gear with G_i disks to G_{i+1} disks; a downshift, switching from a gear with G_i disks to G_{i-1} disks. A gear switch can be either an up-shift or a downshift.

4.2 Update Propagation

When a powered-off disk misses a write request, it must synchronize the stale data either when powered on or just before the stale information is accessed. If there is a lot of stale data, fully synchronizing a disk can be slow. The on-demand approach updates stale data only when it is accessed, allowing the gear shift to take place much more swiftly, but the full-synchronization approach is simpler to build. The on-demand approach is not applicable for downshifts, since PARAID needs to finish the propagation before spinning down drives.

The disk manager captures outstanding writes to powered-off disks. For full synchronization, the disk manager reissues outstanding writes to the disk when it is powered on, possibly rereading some data from replicated soft states stored in the current gear.

For on-demand synchronization, the PARAID block I/O handler uses a dirty-block list. If a referenced dirty block is not cached, PARAID will retrieve the block from the original gear and return it to the requestor. PARAID will then write that block to the target-gear disks, effectively piggybacking the synchronization step at access time.

The disk manager must track stale block locations for synchronization. This list of dirty blocks is stored on disk in case of system failure and in memory for fast access.

A failed disk can stop the gear-shifting process. Disks can also fail during synchronization. However, the list of outstanding writes is maintained throughout the disk failure and recovery process. Once the failed disk recovers, the synchronization can resume.

The choice of on-demand or full synchronization for up-shifting is configurable. On-demand allows PARAID to be more responsive to sudden request bursts, at the cost of tracking additional writes for unsynchronized disks. The full-synchronization approach may be preferable for few gear shifts and a read-dominated workload, since the number of blocks to be synchronized is small. The full synchronization method is also available for manual maintenance, such as when an administrator would need to have a consistent system state before pulling out a disk.

4.3 Asymmetric Gear-Shifting Policies

The disk manager performs shifts between gears. The PARAID monitor decides when a shift is needed, and the disk manager then performs the actual power cycles.

Switching to a higher gear is aggressive, so that the PARAID device can respond quickly to a sharp and sustained increase in workload. However, the algorithm should be resilient to short bursts, or it will lead to little energy savings. Downshifting needs to be done conservatively, so that wild swings in system activity will not (1) mislead the PARAID device into a gear that cannot handle the requests, or (2) cause rapid oscillations between gears and significantly shorten the life expectancy of disks.

Up-shifts: To decide when to up-shift, the monitor must know whether the current gear has reached a predetermined utilization threshold, in terms of busy RAID milliseconds within a time window. Interestingly, we could not check the disk-busy status directly, since this probe would spin up a powered-down disk. Instead, an active RAID device is marked busy from the point when a request enters the RAID queue to when the completion callback function is invoked. Since multiple RAID requests can overlap, should a request be completed with an elapsed time of *t* milliseconds, we mark the prior *t* milliseconds busy.

The threshold and time window are configurable, and are set to 80% (based on prior studies [CARR03]) and 32 seconds (based on empirical experience). The intent is that within the time it takes to spin up the disk and propagate updates, the utilization threshold will not reach 100%. The use of an online algorithm to set thresholds automatically will be future work.

To track the system load, the monitor keeps a moving average of utilization $0 \leq U \leq 1$ for each gear. The purpose of averaging is to filter out short bursts of requests that are frequently seen in real-world workloads. The monitor also keeps a moving standard deviation S. If the utilization plus its standard deviation exceeds the threshold $0 \leq T \leq 1$, an up-shift is performed.

$$U + S > T \qquad \text{(Up-shift condition)}$$

The addition of standard deviation makes up-shift more aggressive; however, since both the moving average and the standard deviation lag behind the actual load, the policy is more responsive to changes that lead to sustained activities.

Downshifts: To decide when to downshift, the utilization of the lower gear $0 \leq U' \leq 1$ needs to be computed, with associated moving standard deviation S'. If their sum is below the threshold T, the lower gear can now handle the resulting load, with associated fluctuations.

$$U' + S' < T \qquad \text{(Downshift condition)}$$

A complication arises when each gear is stored in RAID with parity blocks. Suppose gear 2 contains a 5-disk RAID-5, and the 5th disk is replicated in gear 1 with a 4-disk RAID-5. After a downshift (i.e. spinning down the 5th disk), a write disk request within PARAID will have a 20% chance of accessing the spun-down disk, resulting in a parity update for gear 2, and another parity update for gear 1. Therefore, to compute the downshift threshold, the monitor must track recent write activity and inflate the percentage of write accesses A_{write} to the to-be-spun-down disk(s) by a weight W of 1.5x (specific to RAID-5, where writes to 1 data block and 1 parity block can be increased to 1 data block and 2 parity blocks). Otherwise, the lower gear will be unable to handle the resulting load, and will shift back up. Therefore, U' is computed with the following formula:

$$U' = U\frac{G_i}{G_{i-1}}\left[A_{read} + A_{write}\left(\frac{G_{i-1}}{G_i} + \frac{G_i - G_{i-1}}{G_i}W\right)\right]$$

4.4 Reliability

The reliability manager rations power cycles and exchanges the roles of gear memberships to prolong the life expectancy of the entire PARAID. The reliability manager is also responsible for recovering a PARAID device upon disk failure. When PARAID fails at the highest gear, the recovery is performed by the RAID of the highest gear. When PARAID fails in other gears, the recovery is first performed by the lowest gear containing the failed disk, since the parity computed for disks in that gear is sufficient to recover the soft states stored on the failed disk. The recovered soft-state data then is propagated to the next higher gear before recovering that gear, and so on. In the worst case, the number of bytes needing to be recovered for a single drive failure is the size of a single disk.

Although PARAID may take much longer to recover in the worst case due to cascaded recoveries, the average recovery time can be potentially reduced by recovering only modified content in the intermediary gears and frequent switching to the highest gears. To illustrate, should a PARAID host read-only content, recovery only involves switching to the highest gear and performing the recovery with the underlying RAID once, since no cascaded update propagations are needed. With modified content, PARAID can selectively recover only the modified stripes and stripes used to recover modified stripes at intermediary gears and propagate them to the highest gear, where a full recovery is performed. Assuming that 2% of disk content is modified per day [KUEN97], and PARAID switches to the highest gear 10 times a day, lightweight cascaded recovery is theoretically possible.

One might argue that PARAID can lengthen the recovery time, and thus reduce the availability of PARAID. On the other hand, PARAID reduces power consumption, and the associated heat reduction can extend drive life by about 1 percent per degree Celsius [HERB06]. Therefore, the tradeoff requires further studies, which is beyond the scope of this paper.

5 Implementation

PARAID was prototyped on Linux (2.6.5), which was chosen for its open source and its software RAID module. The block I/O handler, monitor, disk manager, and reliability manager are built as kernel modules. A PARAID User Administration Tool runs in user space to help manage the PARAID devices. For reliability, data blocks for all gears are protected by the same RAID level. Although we have not implemented drive rotations, our gear-shifting policies and the characteristics of daily work cycles have limited the number of disk power cycles quite well. We have not implemented the mechanisms to recover only modified stripes in intermediary gears to speed up cascaded recovery.

Linux uses the `md` (multiple device) device driver module to build software RAIDs from multiple disks. For the PARAID block handler implementation, we changed the `md` driver to make it PARAID-aware. The data path of the `md` driver is intercepted by the PARAID device layer, so that requests from conventional RAID are redirected to the block queues of PARAID, which remaps and forwards requests to other conventional RAID-device queues.

During initialization, the PARAID-aware `md` module starts a daemon that provides heartbeats to the PARAID device and calls the monitor periodically to decide when to gear-shift. The disk manager controls the power status of disks through the disk device I/O control interface.

As an optimization, to limit the synchronization of content of a powered-off disk only to updated content, the disk manager keeps a per-disk red-black tree of references to outstanding blocks to be updated. This tree is changed whenever an update is made to a clean block on a powered-off disk. The upkeep of this data structure is not CPU-intensive. Currently, the disk manager synchronizes all modified blocks after bringing back powered-off disks, by iterating through the tree for each disk and reissuing all outstanding writes. For each block to be synchronized, the disk manager reads the block from the original gear, and then writes it to the disks being brought back online. Once synchronization is complete, the gear-shifting manager switches to the new gear. Note that the red-black tree is only an optimization. In the case of losing this tree, gear content will be fully propagated. A new tree can be constructed once PARAID gear switches to the highest gear.

Currently, PARAID serves requests from the current gear until the target gear completes synchronization, a conservative method chosen for implementation ease and to assure that no block dependency is violated through update ordering. In the future, we will explore using back pointers [ABDE05] to allow the new gear to be used during update propagation.

For the PARAID monitor, we currently use 32-second time windows to compute moving averages of disk utilization. The choice of this time window is somewhat arbitrary, but it works well for our workloads and can tolerate traffic bursts and dampen the rate of power cycles. Further investigation of the gear-shifting triggering conditions will be future work.

The `mkraid` tool, commonly used by Linux to configure RAIDs, had to be changed to handle making PARAID devices and insertion of entries in `/etc/raidtab`. Additional `raidtab` parameters had to be defined to be able to specify the gears.

PARAID contains 3,392 lines of modified code to the Linux and `Raidtools` source code. Since the PARAID logic is contained mostly in the Linux Software RAID implementation, it should be portable to future Linux kernel versions and software RAID implementations in other operating systems. We inserted four lines into `raid0.c` and `raid5.c` to set a flag to forward the resulting I/O requests to PARAID.

6 Performance Evaluation

Since the study of energy-efficient approaches to RAIDs is relatively recent, most prior work has been done analytically or via simulations. Analytical methods provide a fundamental understanding of systems. Simulation studies allow for the exploration of a vast parameter space to understand system behaviors under a wide range of scenarios. We chose implementation and empirical measurements to see if we could overcome unforeseen physical obstacles and conceptual blind spots to bring us one step closer to a deployable prototype. When we designed, implemented, and evaluated PARAID, we discovered why an empirical study is difficult for systems designed to save energy.

- Prototyping PARAID was the first barrier, since the system had to be stable enough to withstand heavy benchmarking workloads.
- Commercial machines are not designed for energy measurements, so we had to rewire drives, power supplies and probes for power measurements.
- The conceptual behaviors of a system are far from close to its physical behaviors; therefore, we had to adjust our design along the way.
- Most benchmarks and workload generators measure the peak performance of a system at steady state, which is not applicable for measuring energy savings, for which we need to capture daily workload fluctuations.
- For trace replays, since our physical system configuration was largely fixed, we had to try to match different trace environments with our physical environments in terms of the memory size, traffic volume, disk space consumption, and so on.

- Although many research trace replay tools are available, more sophisticated ones tend to involve kernel hooks and specific environments. Incompatibility of kernel versions prevented us from leveraging many research tools.
- Finally, since it cannot be easily automated and cheaply parallelized, measuring energy savings on a server was very time-consuming.

Considering these challenges, we document our experimental settings to obtain our results. We demonstrate the power savings and the performance characteristics of PARAID by replaying a web trace (Section 6.1) and the Cello99 trace [HP06] (Section 7). The web workload contains 98% reads and is representative of a very large class of useful workloads. The Cello99 workload is I/O intensive, and consists of 42% writes. We used the PostMark benchmark [KATC97] (Section 8) to demonstrate PARAID's performance under peak load. To demonstrate that PARAID can reuse different RAID levels, PARAID was configured with RAID-0 for the Web workload, and RAID-5 for the Cello99 workload. The PostMark benchmark stresses the gear-shifting overhead. All experiments were conducted five times. Error curves were removed from graphs for clarity. Generally, the standard deviations are within 5% of the measured values, with the exceptions of latency and bandwidth numbers, which tend to be highly variable.

6.1 Web Trace Replay Framework

The measurement framework consisted of a Windows XP client and a Linux 2.6.5 server. The client performed trace playback and lightweight gathering of measurement results, and the server hosted a web server running on a RAID storage device [FUJI06] (Table and Figure 6.1.1). On the server, one disk was used for bootstrapping, and five disks were used to experiment with different RAIDs. The client and server were connected directly by a CAT-6 crossover cable to avoid interference from extraneous network traffic.

	Server	Client
Processor	Intel Xeon 2.8 Ghz	Intel Pentium 4 2.8 Ghz
Memory	512 Mbytes	1 Gbytes
Network	Gigabit Ethernet	Gigabit Ethernet
Disks [FUJI06]	Fujitsu MAP3367 36.7Gbytes 15K RPM SCSI Ultra 320 8MB on-disk cache 1 disk for booting 5 disks for RAID experiments Power consumption: 9.6 W (active) 6.5 W idle (spinning) 2.9 W standby (spun-down, empirically measured)	Seagate Barracuda ST3160023AS 160 Gbytes 7200 RPM SATA

Table 6.1.1: Hardware specifications.

To measure the power of the disks, we used an Agilent 34970A digital multimeter. Each disk probe was connected to the multimeter on a unique channel, and the multimeter sent averaged data to the client once per second per channel via a universal serial bus.

To measure the power of a disk, we inserted a 0.1-Ω resistor in series in the power-supply line (Figure 6.1.2). The multimeter measured the voltage drop across the resistor, V_r. The current I through the resistor—which is also the current used by the disk—can be calculated as V_r/R. Given the voltage drop across the disk, V_d, its power consumption is then V_d times I.

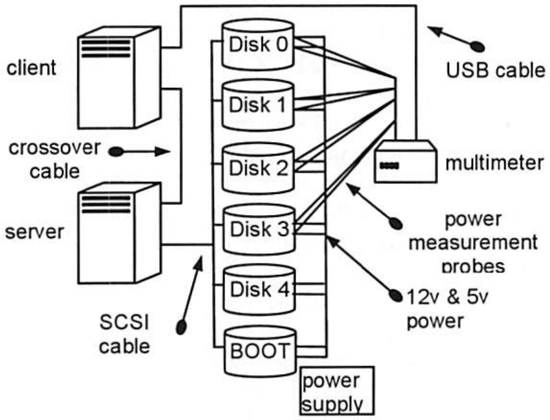


Figure 6.1.1: The measurement framework.

In the measurement system, we removed each disk from the server and introduced a resistor into its +12V and +5V power lines. The +12V line supplied power to the spindle motor; the +5V line provided power to the disk electronics. The SCSI cable was connected directly to the motherboard, allowing the cable to maintain the same performance as if the disks were connected to the SCSI hot swappable backplane in the server.

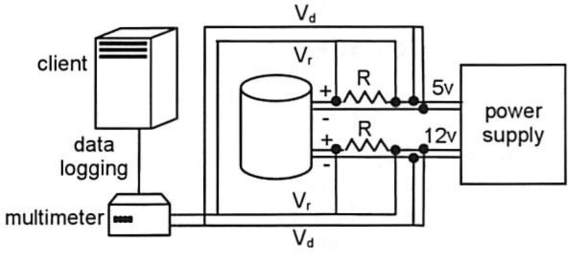


Figure 6.1.2: The resistor inserted in series between the power supply and the disk adapter.

On the client, the Agilent Multimeter software logged the data using Microsoft Excel XP. The multi-threaded trace driver, implemented in Java 1.5, was designed to replay web access log traces and collect performance numbers. Associated requests generated from the same IP address are each handled by a separate thread, to emulate users clicking through web pages. The trace driver also collected server-side and end-to-end performance numbers.

The server hosted an Apache 2.0.52 web server on top of an ext2 file system operating over a RAID storage device that is described in Table 6.1.1.

6.2 Web Server Workload

Workload characteristics affect PARAID's ability to save energy. Under a constant high load, PARAID will not have opportunities to downshift and save energy. Under a constant light load, trivial techniques like turning everything on and off can be used to save energy. In practice, workloads tend to show cyclic fluctuations. The chosen workload needs to capture these fluctuations to demonstrate PARAID's energy savings.

We chose a web server workload from the UCLA Computer Science Department. Since the web content is stored in a decentralized fashion via NFS mounts, we only report the hardware configuration of the top-level web server, which is a Sun Ultra-2 with 256 Mbytes of RAM, 200 Mhz UltraSPARC I CPU, one 2-Gbyte system disk and one 18-Gbyte external SCSI disk, running Apache 1.3.27. Activity was captured from August 10, 2006 to August 16, 2006. Various NFS file systems contained approximately 32 Gbytes of data and ~500K files. We recreated the file system based on the referenced files in the trace. For each full path referenced, every directory in the full path and the file was created according to the order of replay. The file blocks stored on the web server were refilled with random bits. Also, the replay did not include dynamic file content, which accounts for relatively few references in this trace.

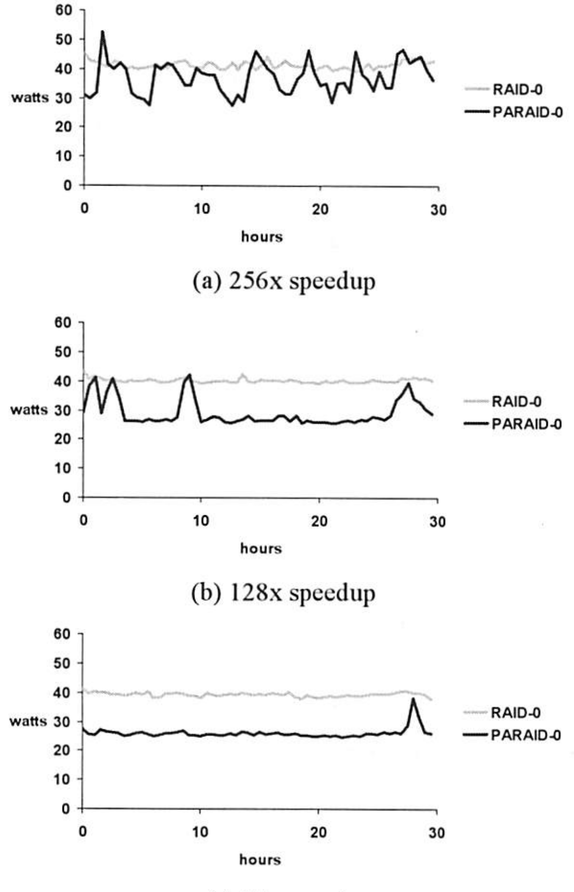
We chose a 30-hour trace starting from 6 PM, August 12, 2006. The duration included 95K requests, with 4.2 Gbytes of data, of which 255 Mbytes are unique. Although the workload is light, it captures the essence of read-mostly cyclic loads and sheds light on PARAID system behaviors, gear-shifting overhead, and the practical implementation limits on power savings.

6.3 Web Trace Replay Experimental Settings

PARAID was compared with a RAID-0 device. The PARAID device used 5 disks, with 2 disks in gear 1, and 5 disks in gear 2. Both client and server were rebooted before each experiment, and PARAID was configured to start with the lowest gear, with gear content pre-populated. The client replayed trace log entries to the server. Due to the hardware mismatch and light trace workload, the collected trace was accelerated at different speeds to illustrate the range of possible savings with different levels of workloads. Experiments included a 256x speedup, which is close to a zero-think-time model, translating to 241 requests/second. With this reference point, we lowered the speedup factor to 128x and 64x, which correspond to 121 and 60 requests/second. All three loads offer few opportunities for the entire 5-disk RAID to be power-switched as a whole. Timing dependent on human interactions, such as the time between user mouse clicks on links (i.e. reference intervals by the same IP) was not accelerated.

6.4 Power Savings

Figure 6.4.1 compares the power consumption of PARAID and RAID-0. Due to the effects of averaging, power spikes are not visible.



(a) 256x speedup

(b) 128x speedup

(c) 64x speedup

Figure 6.4.1: Power consumption for web replay.

PARAID demonstrates a 34% overall savings (ratios of areas under the curves) at 64x. The results approximately match the 33 - 42% range based on equation (2), indicating that further load reduction will yield limited energy benefits. However, turning off 3 out of 5 drives achieves nowhere near 60% energy savings, for PARAID or other RAID systems that save power by spinning down disks. Powering off a disk only stopped it from spinning its platter and therefore only the 12V line was shut off. Power was still needed for the 5V line that powered the electronics, so that it could listen for a power-up command and pass commands along the daisy-chained SCSI cable.

Based on our measurements, spinning up a disk can consume 20-24W. Also, a spun-down disk still consumes 2.9W, noticeably higher than the 1.0W to 2.5W extracted from various datasheets and used in many simulations [GURU03, HUAN03, PINH04, ZHU04, ZHU04B, ZHU05]. The results show that variations in physical characteristics can change the expected energy savings significantly. In our case, if we replace our Fujitsu [FUJI06] with the commonly cited IBM Ultrastar 36Z15 [IBM06], we anticipate an additional 5% energy savings.

The second observation is that the traffic pattern observed in the web log does not correlate well with the disk power consumption. Although this finding reveals more about the nature of caching than the energy benefits of PARAID, it does suggest the value of further investigations into the relationship between server-level activities and after-cache device-level activities.

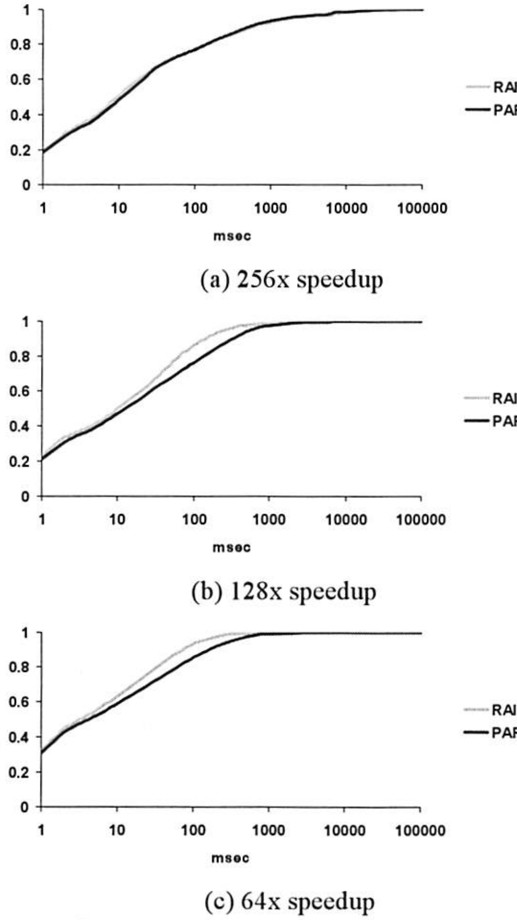
Table 6.4.1 summarizes the overall PARAID energy savings.

Speed-up	Power savings
256x (241 req/sec)	10%
128x (121 req/sec)	28%
64x (60 req/sec)	34%

Table 6.4.1: Percent energy saved for web replay.

6.5 Performance

Latency: Figure 6.5.3 shows the CDFs of per-request latency, which measures the time from the last byte of the request sent from the client to the first byte of data received at the client.



(a) 256x speedup

(b) 128x speedup

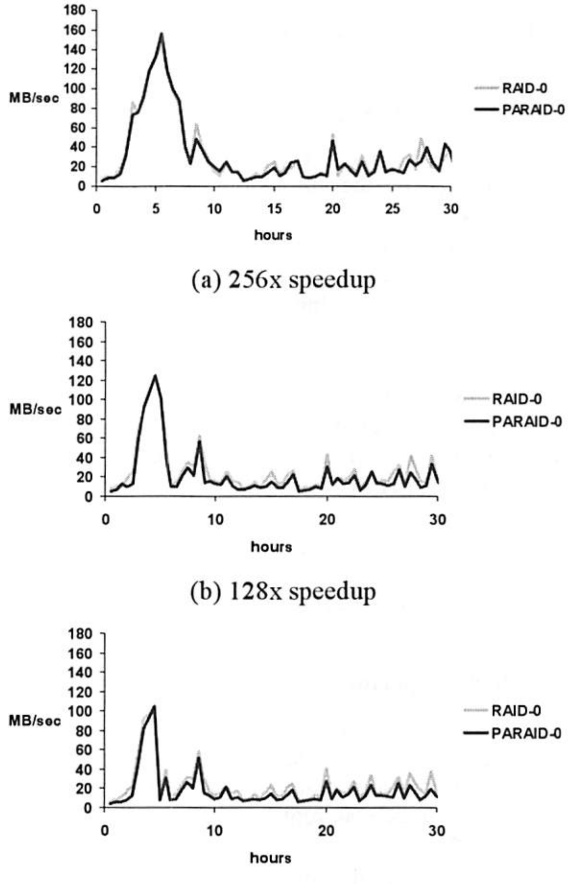
(c) 64x speedup

Figure 6.5.3: Latency for web replay.

As expected, when playing back the trace at high speed, PARAID spent more time at the high gear and used the original RAID-5 disk layout, and the latency CDFs matched closely. The average latency is within 2.7% (~840ms). The data path overhead of PARAID is negligible (Section 8).

When the load was light at 64x, PARAID spent most of the time at the lower gear. PARAID-0 had to use 2 disks to consolidate requests for 5 disks. As a result, the average latency PARAID-0 was 80ms compared to 33ms of RAID-0. However, a web end user should not notice the response time difference during light loads.

Bandwidth: Figure 6.5.4 shows the bandwidth over time, which measures the number of bytes transferred in a 30-minute interval, divided by the time the client spent waiting for any request to complete within the same interval.



(a) 256x speedup

(b) 128x speedup

(c) 64x speedup

Figure 6.5.4: Bandwidth for web replay.

As expected, when PARAID operates mostly in low gear, having fewer active disks leads to lower bandwidth numbers during light loads (Figure 6.5.4 (c)), 24 MB/sec as opposed to 31 MB/sec for RAID-0. However, during the time intervals when PARAID operates in high gear, the peak load bandwidth matches well with the original RAID (within 1.3% of 32MB/sec).

Note that due to time-based data alignment and averaging effects, Figure 6.5.4 (a) only shows a close bandwidth match when PARAID's high-gear performance dominates within a time bracket. Section 7 will also explore request-based alignment to demonstrate bandwidth matching.

Gear-switching statistics: Table 6.5.1 summarizes various PARAID gear-switching statistics for the web replay experiment. Clearly, PARAID spends more time in the low gear as the intensity of workload decreases with the replay speed. Also, each gear switch introduces up to 0.1% extra system I/Os. Interestingly, frequent gear switches can reduce the per-switch cost down to 0.041%, since less time is available for updates to accumulate at a given gear.

	256x	128x	64x
Number of gear switches	15.2	8.0	2.0
% time spent in low gear	52%	88%	98%
% extra I/Os for update propagations	0.63%	0.37%	0.21%

Table 6.5.1: PARAID gear-switching statistics for web replay.

7 HP Cello99 Replay

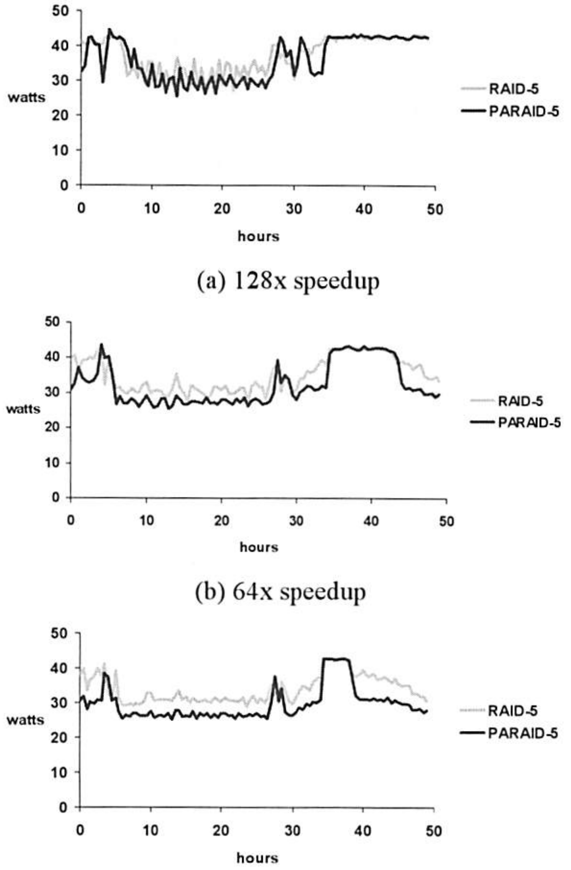
The HP Cello99 trace [HP06] is a SCSI-controller-level trace collected by the HP Storage Research Lab from January 14 to December 31, 1999. The Cello99 data represents IO-intensive activity with writes, which is in contrast to the read-mostly UCLA web with lighter traffic. The traced machine had 4 PA-RISC CPUs, and some devices are md devices, so we had to extract a trace that neither overwhelms our system nor produces too little traffic. The `spc` formatted trace file was generated from the Cello99 data using `SRT2txt`, a program that comes with the HP Cello99 data. The generated trace file was further trimmed so that only the activity associated with `lun 2` was used. Also, we looked for traces with cyclic behaviors. The extracted trace contains 50 hours beginning on September 12, 1999, consisting of ~1.5M requests, totaling 12 GB (stored in 110K unique blocks).

7.1 Cello99 Experimental Settings

PARAID was compared this time with a RAID-5 device. We used a 3-disk gear and 5-disk gear, each reusing the RAID-5 disk layout and reliability mechanisms. The Cello99 trace was replayed on the server at 128x, 64x, and 32x speedup factors to vary the intensity of workloads, corresponding to 1020, 548, and 274 requests/second. The energy measurement framework is the same as depicted in Figure 6.1.1. The server was rebooted before each run, with PARAID configured to start in the lowest gear.

7.2 Power Savings

Figure 7.2.1 compares the power consumptions of PARAID and RAID-5. PARAID demonstrates a single- point-in-time savings of 30% at 128x speedup (~13 hours into the replay) and a 13% overall power savings at 32x speedup. Equation (2) suggests a power saving range of 22 - 28%. Adjusted by the time spent at the high gear (no energy savings), PARAID should have saved 17 - 22% at 32x, 14 - 18% at 64x, and 10 - 13% at 128x. Based on Table 7.2.1, PARAID gear switches, update propagations, and the additional parity computation incur about 4 – 10% of energy overhead, a future goal for optimization. Nevertheless, despite the heavy load of 270 – 1000 requests/second, PARAID can still conserve up to 13% of power.



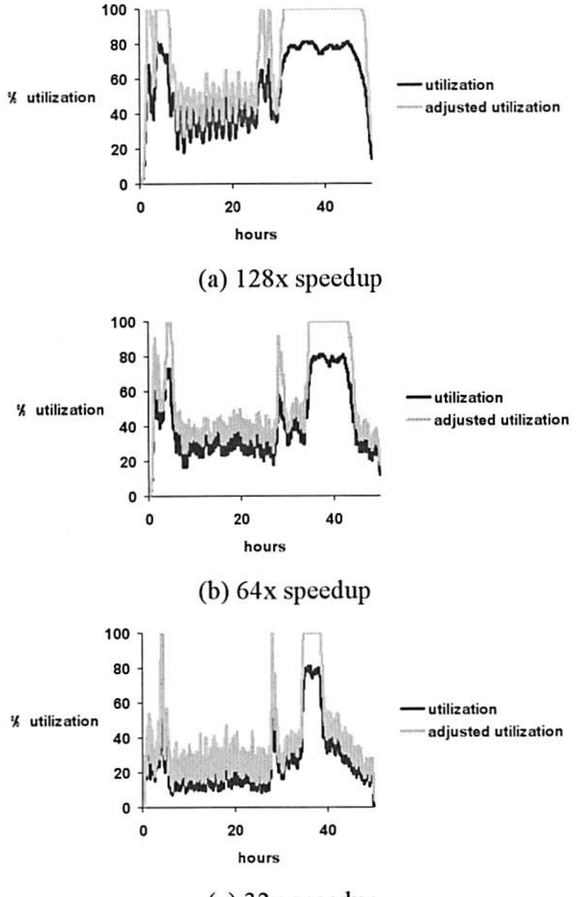
(a) 128x speedup

(b) 64x speedup

(c) 32x speedup

Figure 7.2.1: Power consumption for Cello99.

Figure 7.2.2 shows how gears are shifted based on the current gear utilization, on the percentage of busy gear seconds within a 32-second window, and adjusted utilization, as if the workload is using the low gear. PARAID consolidates the load spread among 5 disks to 3 disks, so that disks 4 and 5 can be spun off, while disks 1 to 3 can operate at 10 – 40% utilization. The graph also reconfirms the lack of opportunities to power-switch the entire RAID for power savings.



(a) 128x speedup

(b) 64x speedup

(c) 32x speedup

Figure 7.2.2: Gear utilization for Cello99 replay. Utilization measures the percentage of busy time of the current gear. Adjusted utilization measures the percentage of busy time of the low gear if the workload is applied to the low gear.

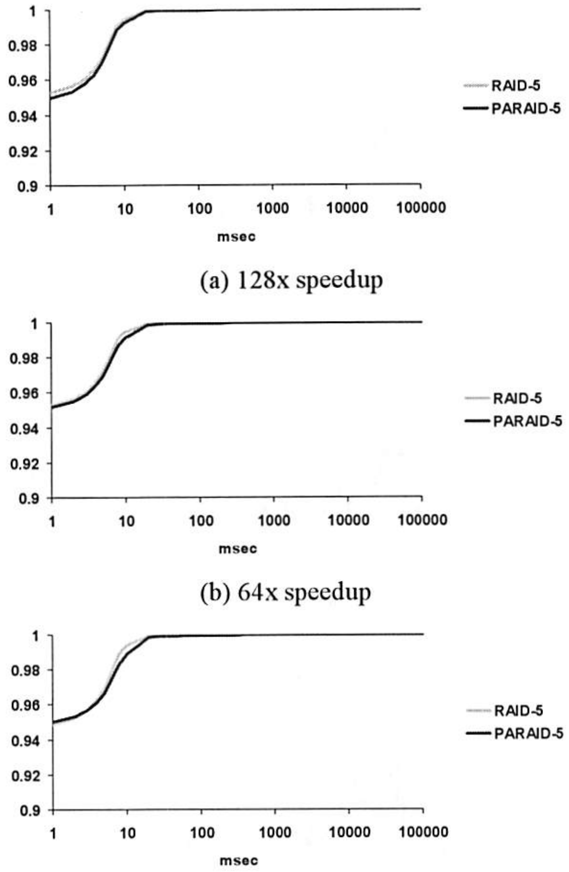
Speed-up	Power savings
128x (1024 req/sec)	3.5%
64x (548 req/sec)	8.2%
32x (274 req/sec)	13%

Table 7.2.1: Percent energy saved for Cello99.

7.3 Performance

Completion time: Figure 7.3.1 shows the CDFs of completion time (from the time of PARAID forwarding a request to the moment the corresponding complete callback function is invoked). Latency is more difficult to measure since blocks are served out of order, and individual blocks from various disks need to be demuxed to the corresponding multi-block request to gather latency information. Therefore, completion time, which is also the worst-case bound for latency, is used.

Unlike the latency CDFs from the web trace, the completion time CDFs of Cello99 showed very similar trends between PARAID and RAID-5, and Figure 7.3.1 presents only the high 90 percentile. At 32x, since PARAID spends more time at the lower gear, its latency is 26% slower than RAID-5 (1.8ms vs. 1.4ms).



(a) 128x speedup

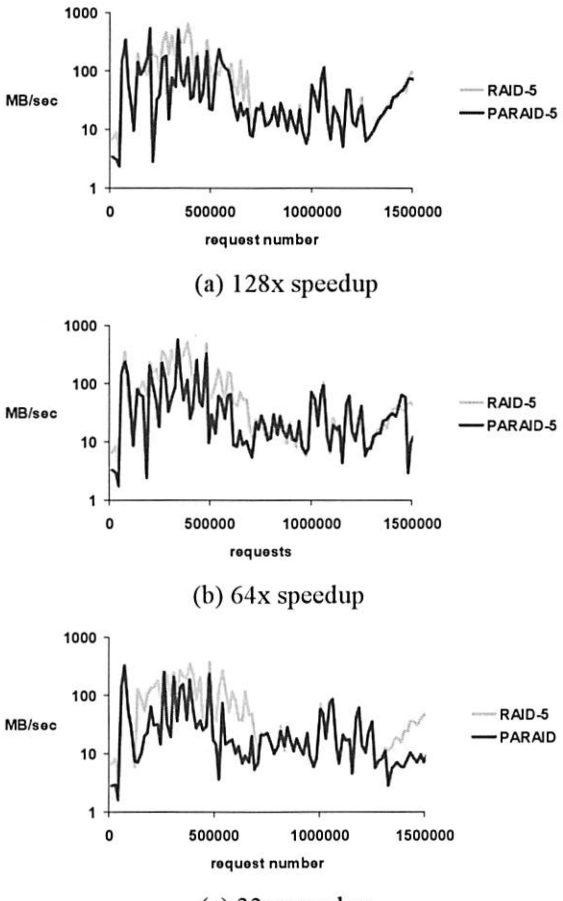
(b) 64x speedup

(c) 32x speedup

Figure 7.3.1: Completion time for Cello99.

We examined the decompositions of I/Os. Although only 51% of bytes are accessed at high gear, they account for 97% of unique bytes. During the light-load periods, such as between the 6th and 27th hour, only 29Mbytes of unique data were referenced. Given that each powered disk can use 5Mbytes of on-disk cache, the bandwidth degradation of PARAID at low gear is significantly dampened by low-level caches. Therefore, the shape of the completion time CDFs is dominated by the high-gear operation, which uses the same RAID-5.

Figure 7.3.2 shows the bandwidth comparisons between PARAID and RAID-5. Note that these graphs are aligned by request numbers to emphasize that 60% of requests that occur during the peak load have the same bandwidth. Whenever PARAID is at high gear, the peak bandwidth is within 1% of RAID-5 (23 MB/sec). The low average bandwidth at high load periods reflects small average request sizes. During periods of light loads, the high bandwidth of both PARAID and RAID-5 reaffirms the enhanced role of low-level caches during light loads. Since PARAID did not use the SCSI controller, which contains additional cache, the bandwidth degradation of PARAID at low gear is likely to be further dampened. When PARAID operates at low replay speed and spends most of its time in the low gear, the average bandwidth degrades as expected (12 MB/sec vs. 21 MB/sec for RAID-5).



(a) 128x speedup

(b) 64x speedup

(c) 32x speedup

Figure 7.3.2: Bandwidth for Cello99.

Gear-switching statistics: Table 7.3.1 summarizes various PARAID gear-switching statistics for the Cello99 replay experiment. Again, PARAID spends more time in the low gear with reduced workload with decreasing playback speed. Due to heavy updates, each gear switch needs to incur an extra 1.3% to 3.9% of system I/Os. Fortunately, gear shifting occurs either before the system becomes highly loaded or is about to downshift due to the upcoming period of light loads. Therefore, these extra I/Os can be effectively absorbed by PARAID with spare I/O capacity, which may otherwise be left unused.

	128x	64x	32x
Number of gear switches	6.0	5.6	5.4
% time spent in low gear	47%	74%	88%
% extra I/Os for update propagations	8.0%	15%	21%

Table 7.3.1: PARAID gear-switching statistics for Cello99.

8 PostMark Benchmark

The PostMark synthetic benchmark generates ISP-style workloads that stress a storage device's peak performance for its read- and write-intensive activity [KATC97]. Running PostMark with PARAID starting at the lowest gear can be indicative of the overhead and latency of gear shifts during a request burst. The PostMark Benchmark was run directly on the server. PARAID propagated updates synchronously during gear shifts.

Figure 8.1 presents PostMark results comparing the elapsed times of RAID 5, PARAID starting with the highest gear, and PARAID starting with the lowest gear under three different benchmark configurations.

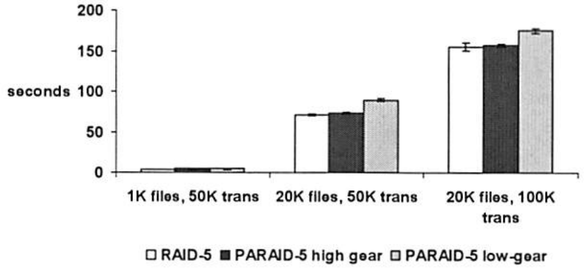


Figure 8.1: PostMark results for a RAID-5 device compared to a PARAID device starting in the highest gear and starting in the lowest gear.

For different PostMark configurations, PARAID starting with the highest gear demonstrates performance similar to RAID 5, which reflects the preserved layout of underlying RAID and minimal disturbances to the md data path. Figure 8.2 shows that as expected, PARAID does not save energy at the highest gear. PARAID energy savings is primarily from low gear settings.

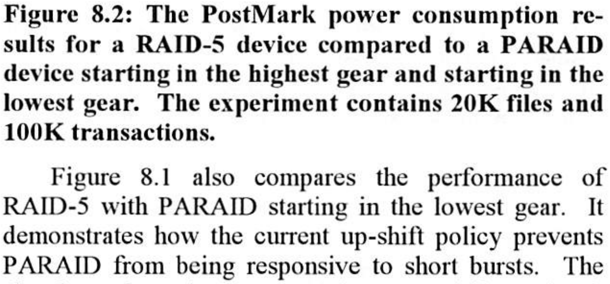


Figure 8.2: The PostMark power consumption results for a RAID-5 device compared to a PARAID device starting in the highest gear and starting in the lowest gear. The experiment contains 20K files and 100K transactions.

Figure 8.1 also compares the performance of RAID-5 with PARAID starting in the lowest gear. It demonstrates how the current up-shift policy prevents PARAID from being responsive to short bursts. The slowdown factor is about 13% due to up-shift overhead. The most responsive approach is to up-shift whenever a burst is encountered. However, this would cause too many gear shifts throughout a day. Our observations suggest that daily cyclic workloads cause few gear shifts, so this overhead is unnoticeable. We plan to explore online algorithms to improve the responsiveness to burst loads while minimizing the number of gear shifts.

Table 8.1 demonstrates that PARAID in either configuration incurs similar CPU and system overhead when compared to RAID-5.

	Mean % CPU	Mean % System
RAID-5	3.24%	41.18%
PARAID high gear	3.11%	41.60%
PARAID low gear	3.08%	41.93%

Table 8.1: PostMark CPU and system overhead for RAID-5, PARAID starting in the highest gear and PARAID starting in the lowest gear. The experiment contains 20K files and 100K transactions.

9 Related Work

Most energy-reduction studies have addressed mobile computing [DOUG95, HELM96]. Recently, energy reduction for servers has also generated interest. Various approaches range from the hardware and RAID levels to the file system and server levels.

Reducing power consumption in hard disks: Carrera, et al. [CARR03] suggest using hypothetical two-speed disks. During periods of high load, the disk runs at maximum speed and power. During periods of light load, the disk spins at a lower speed and possibly idles. They report simulated disk energy savings between 15% to 22% for web servers and proxy servers, with throughput degradation of less than 5%.

FS2 [HUAN05] replicates blocks on a single disk to improve performance and reduce energy consumption via reducing seek time. FS2 reports up to 34% improvement in performance. The computed disk power consumption for per disk access also shows a 71% reduction. Since FS2 does not attempt to spin down disks, and since PARAID has spare storage for disks in high gears due to skewed striping (Figure 3.1.1), FS2 can be used on disks in high gears to extend PARAID's power savings.

Energy-efficient RAIDs: Hibernator [ZHU05] aims to reduce the energy consumption of RAIDs without degrading performance through the use of multi-speed disks. According to demand, data blocks are placed at different tiers of disks spinning at different speeds. A novel disk-block distribution scheme moves disk content among tiers to match disk speeds. When performance guarantees are violated, Hibernator spins disks at full speed to meet the demand. In simulation, Hibernator shows up to 65% energy savings.

Unlike Hibernator, PARAID is designed for existing server-class disks, and the minimum deployment granularity can be a small RAID on a typical server. Also, legacy systems can deploy PARAID via a software patch. As one consequence, some of the PARAID disks running at the lowest gear have few power-saving options. The future ubiquity of multi-speed disks will allow PARAID to explore further energy savings when running at the lowest gear.

MAID [COLA02] assumes that the majority of the data is being kept primarily for archival purposes, and its energy savings are based on the migration of this inactive majority to rarely used disks that fill a role similar to tape archives. PARAID, on the other hand, assumes that all data must be available at a high speed at all times. PARAID's techniques could be used on MAID's relatively few active disks to further improve the performance of that system.

Popular data concentration (PDC) [PINH04] saves energy by observing the relative popularity of data. PDC puts the most popular data on the first disk, the second most popular on the second disk, etc. Disks are powered off in PDC based on an idleness threshold. Without striping, PDC does not exploit disk parallelism.

In the absence of disk striping, the power-aware cache-management policy (PA-LRU) [ZHU04] saves power by caching data blocks from the less active disks. Lengthening the access interval for less active disks allows them to be powered off for longer durations. Partition-based cache-management policy (PB-LRU) [ZHU04B] divides the cache into separate partitions for each disk. Each partition is managed separately by a replacement algorithm such as LRU. PB-LRU provides energy savings of 16%, similar to that of PA-LRU.

EERAID [LI04] and its variant, RIMAC [YAO06], assume the use of a nonvolatile cache at the disk-controller level and the knowledge of cache content to conserve energy in RAIDs. Both lengthen disk idle periods by using nonvolatile disk controller cache to delay writes and computing parity or data-block content on the fly. Both spin down at most one disk for RAID-5, which limits their power savings.

[PINH06] generalizes RIMAC to erasure encoding schemes and demonstrates energy savings up to 61% in simulated tests. This approach defines and separates the primary data from the redundant data and stores them on separate disks. Then, the system makes redundant data unavailable at times to save energy. Writes are buffered via nonvolatile RAM.

Energy-aware storage systems: BlueFS [NIGH05], a distributed file system, uses a flexible cache hierarchy to decide when and where to access data based on the energy characteristics of each device. Through empirical measurements, BlueFS achieved a 55% reduction in file system energy usage. Adding PARAID to BlueFS can improve energy benefits.

The *Conquest-2* file system [XU03] uses nonvolatile RAM to store small files to save energy consumed by disks. PARAID can be readily used as a counterpart to serve large files while conserving energy.

Saving power in server clusters: Chase, et al. [CHAS01] and Pinheiro, et al. [PINH01] have developed methods for energy-conscious server switching to improve the efficiency of server clusters at low request loads. They have reported energy reductions rangin from 29% to 43% for Webserver workloads.

PARAID can be combined with the server paradigm, so that over-provisioned servers used to cushion highly bursty loads or pre-powered to anticipate load increases can turn off many PARAID drives. Since powering on disks is much faster than booting servers, PARAID incurs less latency to respond to traffic bursts.

When traffic loads involve a mixture of reads and writes, disk switching in PARAID provides localized data movement and reduces stress on the network infrastructure. Also, PARAID can be deployed on individual machines without distributed coordination.

Other alternatives: Instead of implementing PARAID, one might use HP AutoRAID's ability to reconfigure to emulate PARAID's behavior [WILK95]. However, one fundamental difference is that reconfiguring a RAID with *D* disks to *D - 1* disks under AutoRAID requires restriping all content stored on *D* disks, while PARAID can restripe the content from a partial stripe, in this case 1 disk.

10 Ongoing Work

PARAID is still a work in progress. First, although PARAID exploits cyclic fluctuations of workload to conserve energy, our experience with workloads suggests that it is difficult to predict the level of benefit based on the traffic volume, the number of requests, the number of unique bytes, the peak-to-trough traffic ratios, and the percentage of reads and writes. We are interested in measuring PARAID with diverse workloads to develop further understandings of PARAID's behavior. Also, we plan to test PARAID with other types of workloads, such as on-line transaction processing [UMAS06].

Currently, PARAID is not optimized. The selection of the number of gears, the number of disks in each gear, and gear-shifting policies are somewhat arbitrary. Since empirical measurement is unsuitable for exploring a large parameter space, we are constructing a PARAID-validated simulation for this purpose, which will allow more exploration of parameters. At the same time, we are investigating analytical approaches to develop online algorithms with provable optimality.

We will modify our disk synchronization scheme to explore asynchronous update propagation, allowing newly powered-on drives to serve requests immediately. We plan to implement selective recovery schemes for intermediary gears to speed up cascaded recovery (Currently, PARAID-5, as used in this paper, recovers 2.7x slower than RAID-5.), and also to incorporate the S.M.A.R.T tools [TOOL05] for disk health monitoring, allowing more informed decisions on rationing power cycles, and rotation of the gear-membership of disks. Finally, we plan to mirror a PARAID server to FSU's department server for live testing, deploy PARAID in a real-world environment, and compare PARAID with other energy-saving alternatives.

11 Lessons Learned

The idea of PARAID was born as a simple concept to mimic the analogy of gear-shifting, which conserves fuel in vehicles. However, turning this concept into a kernel component for practical deployment has been much more difficult than we anticipated.

First, our early design and prototype of PARAID involved cloning and modifying RAID-0. As a result, we had to bear the burden of inventing replication-based reliability mechanisms to match different RAID levels. Our second-generation design can reuse the RAID encoding scheme, making the evolution of new RAID levels independent of PARAID. Although the resulting energy savings and performance characteristics were comparable in both implementations, PARAID's structural complexity, development time, and deployment potential improved in the new design.

Second, measuring energy consumption is difficult because of data-alignment problems and a lack of integrated tools. With continuous logging, aligning data sets is largely manual. For multi-threaded experiments and physical disks, the alignment of data sets near the end of the experiment is significantly poorer than it was at the beginning. Early results obtained from averages were not explicable, since unaligned square waves can be averaged into non-square shapes.

Third, measuring systems under normal loads is harder than under peak loads. Replaying traces as quickly as possible was not an option, and we had to explore different speedup factors to see how PARAID reacts to loads changes. Since server loads have constant streams of requests, we cannot simply skip idle periods [PEEK05], because such opportunities are relatively infrequent. Worse, consolidated workloads are carried by fewer powered-on components with less parallelism, further lengthening the measurement time.

Fourth, modern systems are complex. As modern hardware and operating systems use more complex optimizations, our perception of system behaviors increasingly deviates from their actual behaviors. Memory caching can reduce disk activity, while file systems can increase the burstiness of RAID traffic arrivals due to delayed write-back policies. Disks are powered with spikes of current, making it difficult to compute power consumption as the areas under the spike. Disk drives can still consume a significant amount of power even when they are spun down.

Fifth, matching the trace environment to our benchmarking environment is difficult. If we use a memory size larger than that of the trace machine, we may encounter very light disk activity. The opposite can saturate the disks and achieve no power savings.

Cyclic workload patterns before the cache may poorly reflect the patterns after the cache. Additionally, traces might not have been made using RAIDs, some traces might be old, and the RAID geometry might not match our hardware settings. The base system might have multiple CPUs, which makes it difficult to judge whether a single modern CPU is powerful enough. Although the research community is well aware of these problems, the solutions still seem to be achieved largely by trial and error.

12 Conclusion

PARAID is a storage system designed to save energy for large computing installations that currently rely upon RAID systems to provide fast, reliable data storage. PARAID reuses standard RAID-levels without special hardware, while decreasing their energy use by 34%. Since PARAID is not currently optimized, and since we measured only 5 drives (among which at least 2 are always powered), we believe that an optimized version of PARAID with many disks could achieve significantly more energy savings.

A second important conclusion arises from the research described in this paper. Actual implementation and measurement of energy-savings systems is vital, since many complex factors such as caching policies, memory pressure, buffered writes, file-system-specific disk layouts, disk arm scheduling, and many physical characteristics of disk drives are difficult to capture fully and validate simultaneously using only simulation. Also, implementations need to address compatibility with legacy systems, the use of commodity hardware, and empirical evaluation techniques, all of which are necessary for practical deployments.

Unfortunately, our research also shows that there are considerable challenges to performing such experiments. We overcame several unforeseen difficulties in obtaining our test results, and had to invent techniques to do so. This experience suggests the value of developing standard methods of measuring the energy consumption of computer systems and their components under various conditions. We believe this is another fruitful area for study.

Acknowledgements

We thank our shepherd Keith Smith and the anonymous reviewers for their invaluable insights. Additionally, we thank Jason Flinn, Daniel Peek, Margo Seltzer, Daniel Ellard, Ningning Zhu, HP, and Sun Microsystems for providing accesses to various tools and traces. Cory Fox, Noriel Lu, Sean Toh, Daniel Beech, and Carl Owenby also have contributed to PARAID. This work is sponsored by NSF CNS-0410896. Opinions, findings, and conclusions or recommendations expressed in this document do not necessarily reflect the views of the NSF, FSU, UCLA, Harvey Mudd College, or the U.S. Government.

References

[ABDE05] M. Abd-El-Malek, W.V. Courtright II, C. Cranor, G.R. Ganger, J. Hendricks, A.J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J.D. Strunk, E. Thereska, M. Wachs, J.J. Wylie, URSA Minor: Versatile Cluster-based Storage. *Proceedings of the 4th USENIX Conference on File and Storage Technology*, 2005.

[ASAR05] T. Asaro. An Introduction to Thin Provisioning. *Storage Technology News*, 2005. http://searchstorage.techtarget.com/columnItem/0,294698,sid5_gci1134713,00.html.

[CARR03] E. Carrera, E. Pinheiro, R. Bianchini, Conserving Disk Energy in Network Servers, *Proceedings of the 17th Annucal ACM International Conference on Super Computers*, 2003.

[CHAS01] J. Chase, D. Anderson, P. Thakar, A. Vahdat, R. Doyal, Managing Energy and Server Resources in Hosting Centers, *Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.

[COLA02] D. Colarelli, D. Grunwald, Massive Arrays of Idle Disks For Storage Archives, *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 2002.

[DOUG95] F. Douglis, P. Krishnan, B. Bershad Adaptive Disk Spin-down Policies for Mobile Computers *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing, 1995.*

[FUJI06] Fujitsu, MAP 10K RPM, 2006. http://www.fujitsu.com/us/services/computing/storage/hdd/discontinued/map-10k-rpm.html.

[GRAY05] J. Gray, Keynote Address Greetings from a Filesystem User, *the 4th USENIX Conference on File and Storage Technologies,* 2005.

[GURU03] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, H. Franke, DRPM: Dynamic Speed Control for Power Management in Server Class Disks, *Proceedings of the International Symposium on Computer Architecture*, 2003.

[HELM96] D.P. Helmbold, D.D.E. Long, B. Sherrod, A dynamic disk spin-down technique for mobile computing, *Proceedings of the 2nd Annual International Conference on Mobile Computing and Networking (MobiCon'06), 1996.*

[HERB06] G. Herbst. Hitachi's Drive Temperature Indicator Processor (Drive-TIP) Helps Ensure High Drive Reliability, http://www.hitachigst.com/hdd/technolo/drivetemp/drivetemp.htm, 2006.

[HP06] HP Labs, Tools and Traces, 2006. http://www.hpl.hp.com/research/ssp/software/

[HUAN03] H. Huang, P. Pillai, K.G. Shin, Design and Implementation of Power Aware Virtual Memory, *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003.

[HUAN05] H. Huang, W. Hung, K.G. Shin: FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption. *Proceedings of the 20th Symposium on Operating Systems Principles*, 2005.

[IBM06] IBM, IBM Hard Disk Drive—Ultrastar 36Z15. http://www.hitachigst.com/hdd/ultra/ul36z15.htm.

[IYEN00] A. Iyengar, J. Challenger, D. Dias, P. Dantzig, High-performance Web Site Design Techniques, *IEEE Internet Computing*, 4(2):17–26, March 2000.

[KATC97] J. Katcher, PostMark: A New File System Benchmark, Technical Report TR3022, Network Appliance Inc., 1997

[KUEN97] G.H. Kuenning, G.J. Popek. Automated Hoarding for Mobile Computers. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[LEVI06] M. Levin. Storage Management Disciplines are Declining. *Computer Economics*. 2006. http://www.computereconomics.com/article.cfm?id=1129.

[LI04] D. Li, P. Gu, H. Cai, J. Wang. EERAID: Energy Efficient Redundant and Inexpensive Disk Array. *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.

[MANL98] S. Manley, M. Seltzer, M. Courage, A Self-Scaling and Self-Configuring Benchmark for Web Servers, *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 1998.

[MILL93] E. Miller, R. Katz, An analysis of file migration in a Unix supercomputing environments, *Proceedings of the 1993 USENIX Winter Technical Conference*, 1993.

[MOOR05] J. Moore, J. Chase, P. Ranganathan, R. Sharma, Making Scheduling "Cool": Temperature-Aware Workload Placement in Data Centers, *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[NIGH05] E.B. Nightingale, J. Flinn, Energy-Efficiency and Storage Flexibility in the Blue File System, *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2005.

[PATT88] D.A. Patterson, G. Gibson, RH Katz, A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD International Conference on Management of Data*, 1(3):109-116, June 1988.

[PEEK05] D. Peek, J. Flinn, Drive-Thru: Fast, Accurate Evaluation of Storage Power Management, *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[PINH01] E. Pinheiro, R. Bianchini, E. V. Carrera, T. Heath. Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, 2001.

[PINH04] E. Pinheiro, R. Bianchini, Energy Conservation Techniques for Disk Array-Based Servers, *Proceedings of the 18th Annual ACM International Conference on Supercomputing*, 2004.

[PINH06] E. Pinheiro, R. Bianchini, C. Dubnicki. Exploiting Redundancy to Conserve Energy in Storage Systems. *Proceedings of Sigmetrics/Performance*, 2006.

[RFC01] RFC-3174 - US Secure Hash Algorithm 1, 2001. http://www.faqs.org/rfcs/rfc3174.html

[TOOL05] SANTools, Inc. 2005. http://www.santools.com/smartmon.html

[SANT99] D.S. Santry, M.J. Feeley, N.C. Hutchinson, A.C. Veitch, R.W. Carton, J. Ofir, Deciding when to forget in the Elephant File System, *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 1999.

[UMAS06] UMass Trace Repository, Storage Traces, 2006. http://signl.cs.umass.edu/repository/walk.php?cat=Storage.

[WILK95] J. Wilkes, R. Golding, C. Staelin, T. Sullivan. The HP AutoRAID Hierarchical Storage System. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[XU03] R. Xu, A. Wang, G. Kuenning, P. Reiher, G. Popek, Conquest: Combining Battery-Backed RAM and Threshold-Based Storage Scheme to Conserve Power, *Work in Progress Report, 19th Symposium on Operating Systems Principles (SOSP)*, 2003.

[YAO06] X. Yao, J. Wang. RIMAC: A Novel Redundancy-based Hierarhical Cache Architecture for Energy Efficient, High Performance Storage Systems. *Proceedings of the EuroSys*, 2006.

[YU00] X. Yu, B. Gum, Y. Chen, R. Wang, K. Li, A. Krishnamurthy, T. Anderson, Trading Capacity for Performance in a Disk Array, *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[ZHU04] Q. Zhu, F.M. David, C. Devaraj, Z. Li, Y. Zhou, P. Cao, Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management, *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.

[ZHU04B] Q. Zhu, A. Shanker, Y. Zhou, PB-LRU: A Self-Tuning Power Aware Storage Cache Replacement Algorithm for Conserving Disk Energy, *Proceedings of the 18th Annual ACM International Conference on Supercomputing*, 2004.

[ZHU05] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, J. Wilkes, Hibernator: Helping Disk Arrays Sleep through the Winter, *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

REO: A generic RAID Engine and Optimizer

Deepak Kenchammana–Hosekote*, Dingshan He†, James Lee Hafner*
*IBM Almaden Research Center, †Microsoft
{kencham,hafner}@almaden.ibm.com, dinghe@microsoft.com

Abstract

Present day applications that require reliable data storage use one of five commonly available RAID levels to protect against data loss due to media or disk failures. With a marked rise in the quantity of stored data and no commensurate improvement in disk reliability, a greater variety is becoming necessary to contain costs. Adding new RAID codes to an implementation becomes cost prohibitive since they require significant development, testing and tuning efforts. We suggest a novel solution to this problem: a generic **R**AID **E**ngine and **O**ptimizer (REO). It is generic in that it works for any XOR-based erasure (RAID) code and under any combination of sector or disk failures. REO can systematically deduce a least cost reconstruction strategy for a read to lost pages or for an update strategy for a flush of dirty pages. Using trace driven simulations we show that REO can automatically tune I/O performance to be competitive with existing RAID implementations.

1 Introduction

Until recently, protecting customer data from loss due to media failure and/or device failures meant storing it using one of five RAID levels [32]. To handle higher performance and reliability needs of customers, storage vendors have deployed hierarchical codes like RAID 51. These codes are offered as a result of juggling the inherent risk-reward trade-off from a software engineering standpoint and not out of any merits, whether in storage efficiency or performance. Since these codes can be composed by re-using e.g., hierarchically, the basic RAID set, source code added was minimal. This meant that product marketing needs could be satisfied with low test expense.

There were good reasons why only a few RAID codes were supported in traditional RAID controller implementations (firmware). Firmware complexity grows with every supported RAID code, increasing development and test costs. When firmware becomes a large collection of specific cases it becomes hard to do path length optimizations. From a software maintainability standpoint, a collection of *if... then... else...* code blocks makes firmware readability harder and more prone to bugs. Each roll out of a RAID code potentially requires field upgrades.

Since deploying firmware changes is painful there is a general mindset to avoid it at all costs. However, recent trends in storage technology and customer focus are forcing a re-evaluation. First, no single RAID code satisfies all aspects of data storage. Supporting a variety of RAID codes becomes valuable for effective information lifecyle management where data should be stored at performance, reliability and efficiency levels that are proportionate to its business value. Second, the nature of reference data is that while the dataset grows from gigabytes to petabytes its reliability must remain relatively constant. Using the same RAID code for all sizes is not practical since disk failures grow with capacity[8, 36]. A third reason is the growing popularity of modular systems where bricks [17] are the building blocks to systems that scale in capacity and performance [36, 31, 2, 37]. Some of these systems [11] even simplify management using fail-in-place strategies. Another trend is to use low cost serial ATA (SATA) disks in building large systems [26]. SATA disks have hard error rates that are 10x higher than comparable SCSI disks [10, 18] while being 30–50% cheaper. Providing high data reliability using less reliable disks requires a greater variety of RAID codes.

In light of the tension to provide a variety of RAID codes without compromising the quality, performance, and maintainability of the firmware, we can draw up a list of requirements for an ideal solution: (1) It should allow for adding new RAID codes without firmware complexity, (2) It should easily support popular RAID codes e.g., XOR-based ones which can be implemented efficiently in hardware and/or software, (3) It should automatically handles any RAID code related error handling e.g., read error to a failed sector or disk, (4) Since error handling constitutes a large fraction of any RAID implementation, ideally, the solution should fold fault-free and fault-ridden cases into common code paths. (4) It should simplify nested error paths e.g., in the process of reconstructing a lost block due to a previous failure, a new sector or disk failure is discovered. While the successful completion can occur only if the RAID code permits, an ideal solution must figure out automatically how to do reconstructions. (5) It should automatically tune I/O performance by leveraging dynamic state e.g., cached pages. (6) It should offer informal arguments for correctness, if not formally provable.

Our contributions

We present our efforts at building a generic RAID Engine and Optimizer (REO) fits the above requirements. It is generic in that it works for any XOR-based erasure (RAID) codes (including N-way mirroring) and under any combination of sector or disk failures. In a typical deployment REO routines are invoked by the block data cache in the I/O stack to read, write, scrub, rebuild, or migrate data stored on disks using RAID codes. REO can systematically deduce reconstruction and update strategies and execute them. In addition, an online optimizer within REO can select a least cost strategy for every read or write based on the current cache content. This optimizer can be configured to minimize any system level objective e.g., disk I/O or memory bus usage. By parameterizing fault state, REO can eliminate myriads of cases including those involving nested recovery into a single code path. Finally, by building on results that have been formally proved, we can informally argue about the correctness of REO.

This paper is structured into a high level overview of REO (Section 3) followed by detailed description (Sections 4- 6). While these sections focus on read and write operations, Section 7 discusses scrub, rebuild, and migrate. An evaluation of the efficacy of optimization is discussed in Section 8. Some adaptations to future trends in I/O architecture are presented in Section 9.

2 Related work

There has been no dearth of RAID codes proposed until now e.g., EVENODD[3], generalized EVENODD [4], X-code [42], RDP [12], WEAVER [19]. Recently, there have been a few non-XOR code implementations [8, 33] but these have remained niches since they offer no special advantage over the simplicity of XOR based codes.

One past effort that has focused on providing firmware environments that permit rapid prototyping and evaluation of redundant disk array architectures was RAIDframe[14]. It modularized the basic functions that differentiate RAID architectures — mapping, encoding and caching. Such a decomposition allowed each aspect to be modified independently creating new designs. Array operations were modeled as directed acyclic graphs (DAGs), which specified the architectural dependencies (and execution) between primitives. While it allowed a structure to specify exception handling, RAIDframe lacked any ability to automatically tune performance.

Recently, RAID system-on-chip (SOC) products [27], [6] and [25] have become available. The Aristos SOC, which exemplifies this category, contains an embedded processor, DMA/XOR engines and host and disk interface logic. Since the processor is programmable it is conceivable that they could support a variety of RAID codes. However, the problem with it is that all error paths must be specified as callbacks (much like RAIDframe) which must be written by the developer. Further, it is unclear to us how automated (if at all) the performance tuning is.

3 Overview

REO is a set of routines invoked by a (block or file) data cache when reading or writing data to a RAID coded volume e.g., RAID 1, RAID 5, EVENODD, etc., as shown in Figure 1. In it, applications generate read and write requests to the I/O subsystem that are serviced by a data cache. With write-back caching, application writes are copied to pages in the data cache and marked dirty. At a later time, as determined by the page replacement policy, dirty pages are flushed (written out) to disks. With read caching, when possible, application reads are served out of the data cache. On a read miss, the cache first fetches the data from the disk(s) and then, returns it to the application. Most data caches dynamically partition write-back and read pages to handle a variety of application workloads. In many RAID controllers and filers [23] the write-back cache is protected from unexpected power failure.

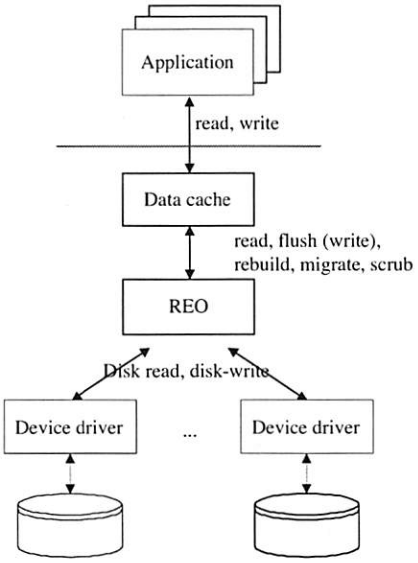


Figure 1: Figure shows a typical deployment of REO within the I/O stack. One or more applications generate read or write calls to RAID coded volumes. These requests are first attempted to be served by the block data cache. If a read miss occurs or a page needs to be flushed then, REO routines are invoked. REO routines include RAID housekeeping functions like rebuild, migrate and scrub. These routines can support any RAID code, under any set of sector or disk failures while simultaneously considering the current cache state.

3.1 Use cases for REO

REO routines are invoked by the data cache in four scenarios:

1. Read on a miss (`reo_read`): The virtualized block address of the requested page(s) is translated to its physical block address within the identified disk. In the fault-free case, disk read(s) is issued. If that disk (or particular sectors within it) has failed then, a reconstruction must be done by REO by reading related blocks according to the RAID code. Sometimes, reconstruction is impossible in which case a read error is returned.
2. Flush a dirty page (`reo_write`): When the cache replacement policy has picked a victim (dirty) page to be written to the disk, the virtual block address of the victim is translated to its physical block address within the identified disk. REO must identify the dependent parity block(s) from the RAID code information and figure out how to update them. This use case covers write-through writes e.g., when write-back caching is disabled.
3. Rebuild a lost page (`reo_rebuild`): Generated by an internal housekeeping routine to repair lost data (due to sector or disk failure), REO must first reconstruct the lost page using the redundant information within the RAID code and then, write it to a new location. Rebuild can be viewed as a composite operation – reconstruct read followed by write.
4. Migrate a page (`reo_migrate`): Triggered by an administrative action, migration is invoked to change the RAID code of a set of pages. Migration includes varying the span (rank) of disks (8-disk RAID 5 to 5-disk RAID 5) and changing the RAID code (RAID 5 to EVENODD). Like rebuild, migration can be viewed as a composition – read, using the old code, followed by write, using the new code.

Since rebuild and migrate are compositions, we focus primarily on describing `reo_read` and `reo_write` operations. We defer discussing `reo_rebuild` and `reo_migrate` to Section 7.

3.2 Two components of REO

REO routines can be functionally partitioned into two components: a RAID Engine which figures out *what* is to be done, and an Execution Engine which figures out *how* it gets done. Figure 2 sketches this breakdown. The RAID engine transforms the input arguments into an I/O plan which comprises a set of blocks to be read, a set to be XOR-ed, and a set to be written. Such an I/O plan is input to the execution engine which issues the necessary disk reads and writes and XOR operations.

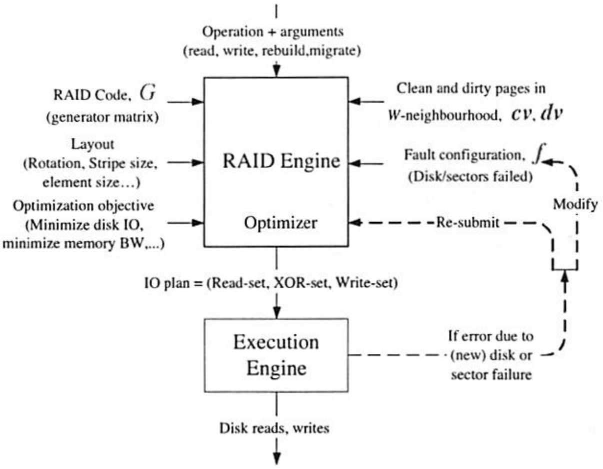


Figure 2: Figure shows a component breakdown of REO into a RAID Engine and Execution Engine. The RAID Engine takes inputs to compute an I/O plan. All its inputs are readily available within the meta-data, system data structures and/or cache directory. An I/O plan includes a set of pages that must be read from the disks, a set of pages that must be XOR-ed, and a set of pages that must be written. Depending on the inputs some of these sets may be empty. The Execution Engine detects and handles error handling during the execution of an I/O plan. If it encounters any errors then, it aborts the I/O plan, modifies the fault configuration vector, and resubmits the operation to the RAID Engine. An online Optimizer within the RAID Engine selects strategies that suit a configured system level objective.

In addition to the basic operation type (`reo_read` or `reo_write`) and their arguments — the page(s) block address, starting virtual block address, and number of bytes to be read or written — the RAID Engine requires the following inputs to generate the I/O plan:

- A concise description of the RAID code, available from the meta-data.
- A description of the physical arrangement of blocks in the RAID code called the layout, available from the meta-data.
- A description of known sector or disk failures called the fault configuration, available from system managed data structures.
- A list of clean and dirty pages presently in the data cache surrounding the page(s) to be read or written, available from the data cache directory.

A final input to the RAID Engine is a resource optimization objective. This can include (but not limited to) criteria like minimizing disk I/O or minimizing memory bus bandwidth. This input guides the Optimizer, a component within the RAID Engine, whenever it has a choice of strategies for any read or write.

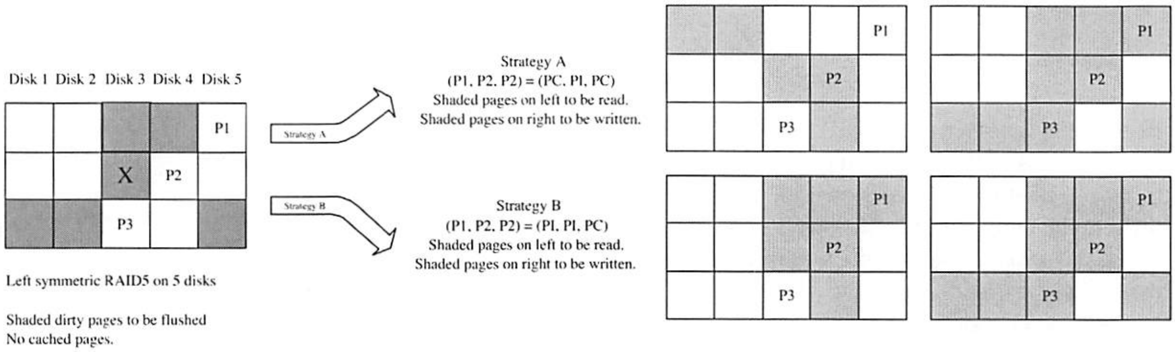

Disk 1 Disk 2 Disk 3 Disk 4 Disk 5
P1
X
P2
P3
Left symmetric RAID5 on 5 disks
Shaded dirty pages to be flushed
No cached pages.
Strategy A
Strategy B
Strategy A
(P1, P2, P2) = (PC, PI, PC)
Shaded pages on left to be read.
Shaded pages on right to be written.
Strategy B
(P1, P2, P2) = (PI, PI, PC)
Shaded pages on left to be read.
Shaded pages on right to be written.
P1
P2
P3
P1
P2
P3
P1
P2
P3
P1
P2
P3

Figure 3: Figure illustrating two (of a total of eight) strategies possible while flushing a set of dirty pages for a 5-disk left-symmetric RAID 5 coded volume. On the left, the shaded pages show the dirty pages that are within a single stripe neighborhood of a victim page marked"X". On the right, we show the resulting pages that would have to be read and written if one strategy were chosen over the other.

The Optimizer has little variety for fault-free reads where the only reasonable option is to read the required page directly from the appropriate disk. For writes there can be more choices. Each choice can require a different number of disk reads and/or XOR operations. In such cases, the configured objective function is used by the optimizer to guide selection.

Before starting on an I/O plan, the Execution Engine acquires all necessary page and stripe locks. Then, it carries out the plan in three phases: first, it submits the disk reads (if any); next, it performs the XOR operations (if any); finally, it submits the disk writes (if any). During any of these steps, if it encounters failed sectors or disks, it re-submits the entire operation to the RAID Engine along with the newly discovered fault state. In Figure 2 shows this resubmission step. One advantage of such a structure is that the recovery code is no different from the main code path.

3.3 Supporting mirrors

One popular RAID code that, at a first glance, appears not to be XOR-based is RAID 1 and, more generally, N-way mirroring. However, such codes are technically degenerate cases of XOR-based erasure codes where each additional copy can be thought of as parity computed from the primary copy and implicit zero entries for a comparable erasure code (RAID 1 with RAID 5, 3-way mirror with EVENODD). Our RAID Engine leverages this to include support for mirrors and striped mirror codes like RAID 10. Without loss of generality, we assume that RAID codes have parity elements in the rest of the paper.

3.4 A RAID 5 example

Before describing the construction of `reo_read` and `reo_write`, we work through an example write that illustrates the choices available and how different I/O plans entail different costs in terms of disk reads and writes and the number of XORs. Figure 3 illustrates this example for a left-symmetric RAID 5 code. In the left figure, say that the dirty pages within a one stripe neighbourhood of the victim page (marked "X") are shaded grey and flushed in a single operation. There are wo strategies possible to perform this operation (shown on right). Each strategy is illustrated by two sets of shaded pages — read pages on left and dirty pages on right. Assuming that a RAID controller can coalesce requests to contiguous blocks on a disk, the approach on the top labelled "Strategy A" requires 4 reads, 6 writes, and 14 pages of memory bus usage for XOR, while "Strategy B" requires 3 reads, 6 writes, and the transfer of 15 pages on the memory bus.

Depending on the configured system level objective REO will choose between these two strategies. Strategy A would be appropriate if memory bus usage were to be minimized; Strategy B is better for disk I/O. The two strategies shown in Figure 3 are from a possible eight.

4 RAID engine

In order to describe the full construction of the RAID Engine we first discuss each of its inputs in greater detail.

4.1 Inputs

4.1.1 RAID code representation

In XOR-based erasure codes (RAID codes) any redundant bit is a XOR of a number of data bits. For efficiency, this relationship is applied to fix-size chunks of bits called *elements*. An element typically consists of one or more consecutive *pages* on disk. Each page is made up of multiple sectors. An element can have either data or parity pages but not a mix of the two. A *stripe* is the set of data elements and all related parity elements. A parity element in a stripe is a XOR of

Disk 1	Disk 2	Disk 3	Disk 4	Disk 5
D1	D3	D5	P1	P3
D2	D4	D6	P2	P4

Figure 4: Physical arrangement of a stripe for a 5-disk 2-fault tolerant EVENODD code. Each strip contains 2 elements. There are 6 data and 4 parity elements in this stripe.

$$\left(\begin{array}{cc|cc|cc|cc|cc}
D1 & D2 & D3 & D4 & D5 & D6 & P1 & P2 & Q1 & Q2 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0
\end{array}\right)$$

Figure 5: Generator matrix G for a 5-disk 2-fault tolerant EVENODD corresponding to its physical arrangement. The vertical lines mark column blocks that correspond to elements within a single strip. Here, the parity arrangement vector is $(7,8,9,10)'$.

some subset of data elements within that stripe. We say that the parity element depends on those data elements. The number of elements that comprise a stripe depends both on the number of disks (called *rank*) and the coding scheme. For example, the 2-fault tolerant EVENODD code over 5 devices has 10 elements in each stripe (Figure 4). Within each stripe, e consecutive elements are arranged contiguously on each storage device forming a *strip*. For simplicity, we assume codes that have uniform sized elements and strips.

The matrix representation of a RAID code is obtained by expressing the XOR relationships between data elements and parity elements as a system of equations [28]. The matrix from such an organization is called its *generator matrix*, G. It is a $N \times M$ binary matrix, where N is the number of data elements in a stripe and M is the total number of elements in a stripe (data and parity). A column of G corresponds to data or parity element in the stripe. A column component of G corresponding to a data element will usually have a single 1. For a parity element, the corresponding column component will have multiple 1's, one for each dependent data element.

If each element is k pages then, G can be rewritten in terms of pages instead of elements by replacing each element entry by an identity matrix of size k. Without loss of generality and for a simpler exposition we assume that each element corresponds to a single page and use the terms elements and pages interchangeably.

4.1.2 Layout representation

Layout is the physical (on disk) arrangement of data and parity pages within a stripe. Besides configuration parameters like the size of each page, much of the layout can be discerned from the generator matrix G for the RAID code and e, the number of pages per strip. As mentioned in the previous section, G can be visualized as blocks of e columns, each block corresponding to physical arrangement of a strip on the disk. When parity pages are interspersed with data pages the layout is interleaved. An example of an RAID code with interleaved layout is the X-Code proposed by Xu and Bruck [42] (Figure 8). Examples of codes with non-interleaved layouts include RAID 5 and EVENODD.

For convenience, it is worthwhile to summarize the location of parity pages within a stripe in a vector of column indicies corresponding to parity pages in G. We call this vector the parity arrangement vector of dimension $1 \times (M-N)$.

To allow for even distribution of load across all disks many layouts are cyclically shifted i.e., columns of the basic codeword are rotated distributing the parity elements evenly on all disks. This shifting can be represented by a signed number s that defines the cyclical shift of strips per stripe. The sign encodes the shift direction - negative for left-symmetric and positive for right-symmetric. Some layouts have no cyclical shifting an example of which is the WEAVER family described in Hafner [19] (Figure 8).

4.1.3 Fault representation

The failure state of a page can be derived from two sources - failure state of the disks and the bad sector table. Both kinds of failure might be either discovered or obtained from system meta-data. We encode the failure state of a set of n pages as the fault configuration vector $\mathbf{f}$ of dimension $1 \times n$, where an entry for page i is marked 1 if that page has failed, otherwise 0. The fault configuration vector gets modified if new errors are discovered in while executing an I/O plan. This is shown in Figure 2.

4.1.4 Cache representation

In a write-back cache, the victim (dirty) page is determined by its replacement policy. While flushing the victim it is efficient to simultaneously flush dirty pages that belong to the same stripe [38]. In REO, we extend this idea by defining a W-neighborhood for a victim page. This is defined as the set of all pages, clean or dirty, in the data cache that are in a $2W+1$ stripe window centered around the victim's stripe. This is shown in Figure 6. By choosing $W > 0$, REO can batch flush requests of multiple pages thereby improving the throughput of the disks.

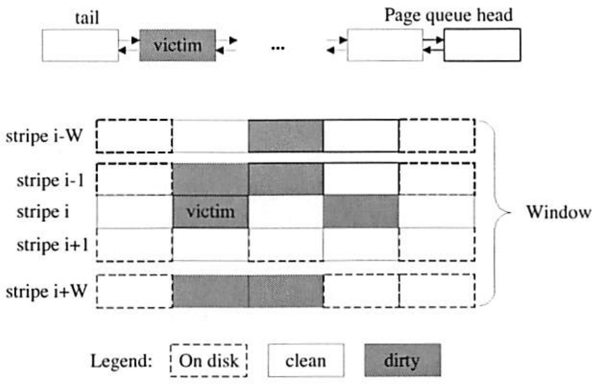


Figure 6: Figure sketches a W-neighborhood of a victim page chosen by the cache replacement policy based on some page list (shown at the top). All dirty pages within this $2W+1$ stripe window (centered around the victim's stripe) are written collectively at a time. Presence of clean pages within the window are leveraged to reduce I/O or XOR.

The set of pages in the W-neighborhood of a victim page can be partitioned into clean and dirty page sets. Each set can be encoded as a binary vector, with a 1 denoting the page in cache. We denote the two vectors - the clean data vector **cv** and the dirty data vector **dv**.

For write-through operations 0-neighborhood is used.

4.2 I/O plan output

As Figure 2 shows, an I/O plan is output by the RAID Engine based on the inputs we have described. Formally, an I/O plan is a 3-tuple $(\mathbf{r}, X, \mathbf{w})$. **r** is a binary vector encoding the set of disk read operations necessary; 1 denoting that that page needs to be read. Similarly, **w** is a binary vector encoding the set of disk write operations necessary. X is the set of XOR operations, each of which is a list of pages to be XOR-ed giving a resultant data or parity page. X can be encoded as a square matrix of dimension $M \times M$ where a column component i describes the set of pages to be XOR-ed to compute parity page i.

4.3 REO read

If the pages needed can be read from good disk(s) then, it is trivial to set **r** to the corresponding pages on those disk(s). In this case both X and **w** are zero.

The challenging case for read is when reconstruction is needed due to sector or disk failures. To derive a reconstruction strategy we employ the scheme described by Hafner et al. [7]. For completeness, we summarize their technique. Starting from the generator matrix G, a modified matrix $\hat{G}$ is derived as follows: for every failed sector, the entries corresponding to that column in G are zeroed; for every failed disk, the columns corresponding to pages on disk are zeroed. Formally, $\hat{G}$ is computed as follows, $\hat{G} = G(I_M - \mathbf{diag}(\mathbf{f}))$, where I_M is the identity matrix of size M and $\mathbf{diag}(\mathbf{f})$ is the matrix derived by applying the fault configuration vector **f** as the diagonal of the $M \times M$ matrix.

Next, using a variant of Gaussian elimination, a pseudo-inverse R ($\hat{G}^{-1}$) is computed. R is of dimension $M \times N$ where the column component i corresponds to a description of the set of surviving pages (data or parity) that must be read and XOR-ed to reconstruct data page i.

Two aspects of this scheme, both of which are discussed and proved by Hafner et al. [7], are central to the RAID Engine's construction. The first is a result that shows that the pseudo-inverse technique will always find, if the RAID code permits, a reconstruction scheme using only the surviving pages (Theorem 1 in that reference). The second aspect is the non-uniqueness of R. From linear algebra, since $\hat{G}$ describes an over-specified system of equations, its inverse will not be unique. Each pseudo-inverse of $\hat{G}$ defines a read strategy. Given a resource optimization objective, an online optimizer can pick a suitable strategy and its relevant pseudo-inverse R. Column components of R that correspond to lost and required pages are extracted to **r** and X. Since some of the required pages might already be in cache, **r** should be logically AND-ed with the clean cache vector **cv** to yield the set of pages that the Execution Engine must read from disk. Note that in the case of reconstruct reads, **w** is zero.

4.4 REO write

4.4.1 Identifying affected parity pages

Recollect that the victim page to be written out is expanded to include the W-neighborhood of dirty pages. The dirty data vector **dv** is the set of pages to be written out including the victim. In any RAID code the changed content of data pages must be reflected to its dependent parity pages. Consequently, the first step is to identify all dependent parity pages. This can be determined by logically AND-ing the dirty data vector with each column component of a parity page in $\hat{G}$. Every resultant non-zero vector implies that the surviving parity page must be updated as part of writing **dv**. Parity pages with resulting zero column implies either that the parity page is unaffected or that the parity page cannot be written because of a sector or disk failure.

In this step we encode the list of affected parity pages as a binary vector where an entry for a parity page is set to 1 if that page is affected, 0 otherwise. We denote this as the affected parity vector.

4.4.2 Selecting a write strategy

The next step is to pick how each affected parity page is to be updated. In RAID 5 any parity element can be updated using one of two approaches — *parity increment* (PI) or *parity compute* (PC). With PI (a.k.a. read-modify-write), the RAID controller first reads on-disk versions of the modified data and parity pages; computes the parity difference between the new and old version of the data page and applies this increment (delta parity) to the old parity to compute the new parity. In PC, it reads all unmodified data pages from disk that a parity page depends on, XORs them with the dirty pages and computes the parity page.

The problem is how to generalize this for any RAID code under any fault configuration. To solve this, first we extend the RAID 5 approach to all RAID codes assuming fault-free configurations and then, we generalize it to allow arbitrary faults.

The extension for a write to a fault-free RAID coded volume is as follows: Any correct update of dirty pages in a RAID coded stripe is some combination of parity increment (PI) or parity compute (PC) for each of the affected parity pages. Note that updates that reuse results from one parity update (a.k.a. delta parity) for another can be re-written in a form showing as if each parity page were updated separately. Each instantiation of a PI or PC for non-zero entries in the affected parity vector defines a write strategy.

This generalization allows one to systematically enumerate all possible write strategies. For any write, if p parity pages are affected then, there will be 2^p write strategies. Since each write strategy translates to a different I/O plan, the optimizer can pick one that best matches the resource optimization objective.

To handle sector or disk failures the above extension is amended to allow reconstruction of the pages needed for PI or PC before the update of the parity page proceeds. Strategy for the necessary reconstruction(s) is chosen using the approach outlined for `reo_read`.

4.4.3 Deriving an I/O plan

Given a write strategy, for each affected parity page, one can compute the pages to be read, XOR-ed and written independently. REO calculates the I/O plan by combining the sub-plans for the affected parity pages. This is done by picking column components from the pseudo-inverse and translating the write strategy — PI or PC — into the necessary reads, XORs and writes. The read set is computed mindful of the clean page vector **cv**.

If the sub-plan for affected parity page k is denoted by $(\mathbf{r}_k, X_k, \mathbf{w}_k)$ then the combined I/O plan is derived by summing all the individual sub-plans.

$$\mathbf{r} = \bigvee_{k \in \textbf{parity}} \mathbf{r_k}; X = \int_{k \in \textbf{parity}} X_k; \mathbf{w} = \bigvee_{k \in \textbf{parity}} \mathbf{w_k}$$

This combined plan is submitted to the Execution Engine.

4.5 An EVENODD example

We work out an example `reo_write` assuming a fault-free configuration of the EVENODD code with a rank of 5 disks. The physical arrangement and generator matrix for this RAID code is shown in Figure 5. This code has two pages per strip ($e = 2$), 6 data pages ($N = 6$) and a total of 10 pages per stripe ($M = 10$) The parity arrangement vector is $(7, 8, 9, 10)'$.

Let's say that there were two dirty pages - D1 and D3. Then,

$$\mathbf{dv} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Say there are no clean pages ($\mathbf{cv} = \mathbf{0}$) and being fault-free ($\mathbf{f} = \mathbf{0}$), $\hat{G} = G(I_M - \mathbf{diag}(\mathbf{f})) = G$.

In the first step, the RAID Engine computes the set of affected parity pages, by AND-ing the dirty page vector with $\hat{G}$ for each parity page. Below is a tabulation of this step for each parity page in the stripe.

P1	P2	Q1	Q2
1	0	1	0
0	0	0	0
1	0	0	1
0	0	0	0
0	0	0	0
0	0	0	0

Notice that in this write, P2 is unaffected. This is inferred by the zero column component.

Step 2 is to pick a write strategy. Say, the RAID Engine picks

$$\mathbf{strategy}_A = \begin{pmatrix} \text{P1} \\ \text{P2} \\ \text{Q1} \\ \text{Q2} \end{pmatrix} = \begin{pmatrix} \text{PC} \\ \text{-} \\ \text{PC} \\ \text{PI} \end{pmatrix} \tag{1}$$

In Step 3 sub-plans are computed for each affected parity page. The sub-plan for P1 is

$$\mathbf{r}_{P1} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} ; X_{P1} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} ; \mathbf{w}_{P1} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The sub-plan for Q1 is

$$\mathbf{r}_{Q1} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} ; X_{Q1} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} ; \mathbf{w}_{Q1} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

The sub-plan for Q2 is

$$\mathbf{r}_{Q2} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} ; X_{Q2} = \begin{pmatrix} 0 \\ 0 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} ; \mathbf{w}_{Q2} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Note that the entry "2" denotes the XOR-ing of the old and new versions of the element.

Finally, the sub-plans are combined to give the I/O plan for this operation:

$$\mathbf{r} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} ; X = \left(\begin{array}{ccc} P1 & Q1 & Q2 \\ \hline 1 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 2 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{array} \right) ; \mathbf{w} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

For brevity, only the non-zero columns of X are shown above. The number atop the horizontal line denotes the column index.

For an example read operation we refer the reader to Section 7 of Hafner et al. [7].

5 Optimizer

In the previous section we discussed how the RAID Engine can enumerate all possible read or write strategies. In this section we discuss the Optimizer's online selection process. Since each strategy translates to an I/O plan, the Optimizer drives selection by defining a measure of goodness for an I/O plan.

5.1 Measures for an I/O plan

For the sake of exposition we describe two plausible measures for a I/O plan.

- The number of distinct disk read and write commands needed to execute an I/O plan. Denoted IOC, this metric is intended to measure disk overhead i.e., total seeks and rotations. In its simplest version, this measure may weight all seeks and/or rotations equally. Minimizing IOC leads to lower disk overhead in service requests which effectively improves the throughput of the disk.
- The number of cache pages input to and output from XOR operations in executing an I/O plan. Denoted XOR, this measures the memory overhead incurred. Minimizing XOR leads to lower memory bandwidth usage.

Note that alternate measures are possible. For example another metric could use variable seek and rotational costs. Yet another could use a measure of disk queue lengths. By using IOC and XOR our intent is to build a framework within which an Optimizer could be built around an objective function appropriate to the deployment scenario.

5.2 Costing an I/O plan

Given IOC and XOR as plausible metrics, the Optimizer can guide selection by costing plans from competing strategies. We describe how the Optimizer can computes these metrics from $\mathbf{r}, X$, and $\mathbf{w}$.

To compute IOC, both $\mathbf{r}$ and $\mathbf{w}$ are interpreted as being blocked. In the resulting vectors, a count of the number of vertical runs of non-zero entries within each block is IOC since a vertical run of non-zero entries can be submitted as one sequential I/O.

To compute XOR, simply sum up all elements in X and the affected parity vector.

In the example in Section 4.5, IOC = 7 and XOR = 13 for the resultant plan.

If the RAID Engine had chosen the following strategy instead of the one in Equation 1,

$$\textbf{strategy}_B = \begin{pmatrix} \text{P1} \\ \text{P2} \\ \text{Q1} \\ \text{Q2} \end{pmatrix} n = \begin{pmatrix} \text{PC} \\ \text{-} \\ \text{PI} \\ \text{PI} \end{pmatrix} \qquad (2)$$

then, IOC = 8 while XOR = 12.

5.3 Selecting strategy

For a given operation, having defined the space of all possible strategies and some metric for any I/O plan, the Optimizer can employ any well-known search technique e.g., exhaustive search, dynamic programming, greedy, randomization, simulated annealing, to find the least-cost plan. The one constraint in selecting a technique is that it must be amenable to online computation. Since strategy selection is done for each I/O that requires doing disk reads and writes, some time spent selecting is acceptable. However, the time spent searching should be well worth the resulting savings in disk I/O.

5.3.1 Optimal approach

Technically, it is possible to exhaustively search for the least cost I/O plan. For reconstruct reads, each distinct pseudo-inverse leads to a strategy. For each strategy the metric for a resulting I/O plan can be computed. The number of distinct pseudo-inverses is exponential in the dimension of its null space and therefore impractical to enumerate for an implementation. A more practical heuristic is described in Hafner et. al. [7] which computes pseudo-inverses that are sparsest which potentially means least IOC or XOR.

For writes, an exhaustive enumeration of all strategies for a given operation has exponential complexity w.r.t. the total number of affected parity pages. For a given operation, the number of affected parity pages grows if a larger neighborhood is used and/or when higher fault tolerant RAID code is employed. Since exhaustive search can be CPU intensive, more practical heuristics are necessary. We describe two such heuristics - BASELINE and GRADIENT. BASELINE represents existing RAID implementations. We suggest GRADIENT as an effective heuristic among a set of search techniques (mentioned earlier) that we experimented with.

5.3.2 BASELINE heuristic

Most RAID 5 implementations (including Linux `md`[40]) employ a simple majority rule to determine a strategy for a write. If a majority of pages for a stripe are dirty then, PC is chosen. Else, PI is chosen. Under degraded mode, they revert to PI. Similarly, for EVENODD and higher distance codes, thresholding algorithms has been suggested to determine the strategy for an entire stripe. The thresholding employed to select between PI and PC for all affected parity elements in the stripe is typically based on comparing the number of dirty pages within the stripe with a pre-computed table based on rank etc.

Algorithm 1 GRADIENT(**AffectedParityVec**)

```
pv = sort(AffectedParityVec) {Sort affected parities in layout order.}
strategy = $\phi$ {Initialize.}
$i = 0$ {Iterate on each affected parity.}
while i < pv.size() do
  if failed(pv[i]) then {Sectors for parity lost.}
    i = i + 1
    continue
  end if
  strategy_A = strategy {Copy strategy so far.}
  strategy_A[i] = PI {If PI for next affected parity.}
  $c_A$ = cost(strategy_A)
  strategy_B = strategy {Copy strategy so far.}
  strategy_B[i] = PC {If PC for next affected parity.}

  $c_B$ = cost(strategy_B)
  if $c_A > c_B$ then {PC is a better option}
    strategy[i] = PC
  else
    strategy[i] = PI
  end if
  i = i + 1 {Move to next affected parity.}
end while
return strategy
```

When dealing with failures, structure within the RAID code can be exploited to minimize recursive reconstructions [21]. The complexity of this generator is $O(p)$ (p is the number of affected parity) for RAID 5 and nearly constant for thresholding schemes. In all implementations we have examined, a window size of 1 is typical i.e., a 0-neighborhood.

5.3.3 GRADIENT heuristic

The GRADIENT heuristic picks a write strategy by incrementally assigning PI or PC to each non-zero entry in the affected parity vector. As each affected parity is assigned the heuristic favors the assignment that results in a lower cost (based on IOC or XOR).

GRADIENT, outlined in the algorithm below, improves on BASELINE since a strategy for the next affected parity page is chosen based on the strategies assigned to previous parity pages. In the algorithm we have omitted obvious inputs like G, **f**, **cv**, **dv**, etc. The problem with GRADIENT is that, like any gradient method, there is no guarantee that it will find the optimal plan. since it cannot avoid getting stuck in local minimas. While the complexity of this generator is comparable to BASELINE, it does requires invoking the costing routine twice for each affected parity page (lines 11 and 14). GRADIENT uses a (static) configured window size. Finding an efficient heuristic for all layouts under all workloads remains an open problem.

6 Execution engine

The Execution Engine employs well-known techniques in firmware design to execute an I/O plan. We summarize its role for clarity and completeness. As would be needed for higher throughput, multiple I/O plans will run concurrently within the Execution Engine. Each plan gets executed in three phases. In Phase 1, any reads are submitted and completed. In Phase 2, any XORs are calculated. Finally, in Phase 3, any writes are submitted and completed.

Prior to Phase 1, the Execution Engine re-blocks **r** and **w** just as was done in section 5.2. The resultant read and write matrices are used by the Execution Engine to coalesce multiple adjacent disk read/write requests into blocks of sequential I/Os.

6.1 Handling concurrent plans

The Execution Engine must ensure that all three phases are executed as part of a single transaction i.e., there is no interleaving of two concurrently executing IO plans that overlap. RAID controllers must reads, XORs and writes atomically to satisfy the "atomic parity update" requirement [38]. If multiple I/O plans overlap on disk sectors then, the Execution Engine must ensure that a consistent ordering of data and parity is seen by each operation. Any robust solution employs one of two techniques — on-disk log [35] or persistent memory (NVRAM). The latter approach is commonly employed in commercial RAID controllers since most do write-back caching which already requires this. In these implementations, a stripe lock table, kept in persistent memory, maintains the lock state of stripes being touched by concurrent I/O plans. The Execution Engine must acquire all locks for stripes in the W-neighborhood before beginning execution. To avoid deadlocks, all resources necessary to complete an I/O plan must be acquired in advance of the plan's execution and in a well-know order.

6.2 Handling failures

During plan execution, various kinds of errors can occur. Errors that arise out of disk timeout or faulty XOR-ing can be easily retried by the Execution Engine. Handling errors that arise due to discovering a new media or disk failure during a plan execution requires a different tack. In this case, we suggest that the Execution Engine abort the plan and resubmit the entire operation to the RAID engine with an updated fault configuration vector. The RAID engine can compute a new (possibly better) plan that reflects the new fault state. As a side effect, the Execution Engine could update any fault meta-data managed by the system.

This step of unifying the error path with the good path in REO is possible because of the generality with which faults are handled. The elimination of potentially nested recovery paths contributes to its simplicity.

7 Other RAID operations

Besides read and write, all RAID controllers must support rebuild. Rebuild is the operation of reconstructing failed pages within a stripe and writing them to new disk locations. The rate at which rebuild is done is a primary determinant of data availability [15]. In this section we discuss how `reo_rebuild` can be made generic to the RAID code.

Most RAID implementations also support RAID migration, a process of re-laying data that was stored in one layout to another. Migration can include changing stripe size or its rank (5-disk RAID 5 code to 7-disk RAID 5 code) or changing the RAID code itself (5-disk 1-fault tolerant RAID 5 code to 7-disk 2-fault tolerant EVEN-ODD code). We discuss how REO can be used to support arbitrary RAID migrations.

If stripes have lost more elements from media or disk failure(s) then, the RAID code can protect against then, none of these operations can complete successfully. This is due to the inherent limitation of the RAID code and not of REO.

7.1 REO rebuild

Rebuild occurs when there is a sector or disk failure. In the former case, rebuild is typically done for the affected stripe which can be scattered over the volume. In the latter case (disk failure), rebuilds are batched. Within each batch, multiple stripes are rebuilt simultaneously since it translates to sequential disk reads and writes. Keeping a deep queue is essential to speeding up rebuilds [30]. In both cases, the basic logic for `reo_rebuild` remains the same.

A rebuild for a set of lost pages within a stripe is executed in two steps. In Step 1, `reo_read` is executed for the set of lost pages. A read strategy for reconstructing the failed pages is picked by the RAID Engine and the resultant I/O plan is executed by the Execution Engine. In Step 2, an I/O plan with **w** set to the reconstructed pages is submitted to the execution engine. **r** and X for such an I/O plan are zero. Note that the I/O plan in Step 2 is slightly different from one generated for a similar `reo_write` — in some cases, parity pages do not need to be written during rebuild.

In some implementations the entire stripe is read and written out instead of just accessing the minimum pages needed for reconstruction. This is done to detect any lurking sector failures. `reo_rebuild` can be suitably modified to reflect this design decision. Some layouts [29] include spare space for rebuild within the stripe. Such information can be easily captured in the layout input to REO.

7.2 REO migrate

In RAID migration, a volume arranged in a source layout is re-arranged into a target layout. Typically, for space efficiency, this migration is done in-place i.e., the same set of sectors and disks in the source layout are reused for the target layout. Migration proceeds in strides, i.e., multiple stripes. Typically, this multiple is either determined by the lowest common multiple (LCM) of stride sizes of the source and target layouts, or by the stride size of the target layout alone. Sometimes, a staging area on disk is used to store temporary results if cache memory is limited. In both cases, the basic steps within `reo_migrate` are the same.

In a simple version, `reo_migrate` is executed in 2 steps. In Step 1, all data pages from the source layout are read using `reo_read`. Any reconstructions, if needed, are done in this step. In Step 2, the list of data pages for the target layout is input to `reo_write` as if all the parity pages were lost. Given this input, a good heuristic will invariably pick PC for all affected parity elements.

In an alternate implementation of `reo_migrate`, all pages (data and parity) in the source layout are read in Step 1. In Step 2, if possible, parity pages from this layout are reused as partial results for computing parity pages of the target page. Reusing parity pages from the source layout has the potential to reduce the memory bandwidth needed for the migration. However, it has the disadvantage that errors that have crept in due to bad sectors get propagated to the target layout.

7.3 Sundry operations

Two other operations commonly implemented are initialization and scrubbing. Initialization is layout independent in that all regions of a volume must be zero-ed. This is typically done in batches by writing sufficiently large writes with zeroes.

Scrubbing is a periodic scan of every stripe to check for latent hard errors. In REO, scrubbing is implemented by using a parity check matrix H for a RAID code which is computed from its generator matrix G. H is a $M \times (M-N)$ matrix where each column component corresponds to a parity page in the RAID code. If all pages with entry 1 in that column are XOR-ed then, the result must be a zero page. A resultant non-zero page implies an inconsistent parity. H is derived from G by rearranging column blocks of G into row blocks in H and including an identity matrix (corresponding to the parity pages). Figure 7 shows an example parity check matrix for the 5-disk 2-fault tolerant EVENODD code.

$$H = \left(\begin{array}{cccc}
P1 & P2 & Q1 & Q2 \\
\hline
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
\hline
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 \\
\hline
1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 \\
\hline\hline
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\hline
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{array}\right)$$

Figure 7: Parity check matrix H for a 5-disk 2-fault tolerant EVENODD code (Figure 5). H is used to detect inconsistent pages. H is derived by rearranging column blocks in G into row blocks and including an identity matrix that corresponds to the parity pages.

Rectifying stripes that fail parity check is challenging. With higher fault tolerant codes it is possible to deduce the location of the error. This is not possible with RAID 5. In some deployments silently correcting such errors is unacceptable. In cases where it is acceptable, the parity element is assumed to be wrong and fixed.

8 Evaluation

In this section we report results on aspects of REO that are amenable to quantitative evaluation. We present empirical results from our experience in adding more than a dozen RAID codes into our simulator. The efficacy of the Optimizer for real workloads is shown by trace driven simulations. Other aspects such as correctness can be shown informally leveraging results proved elsewhere [7].

8.1 Versatility

Table 1 lists a representative set of RAID codes that we implemented in a simulator. These codes vary in fault tolerance, physical arrangement, efficiency, and performance. A visual guide to stripes of these codes is shows in Figure 8. To date, we have added more than a dozen RAID codes. Adding a new code meant specifying its generator matrix and its layout, a task that averaged about 15 minutes. We believe that this empirical data can be cautiously extrapolated to real implementations while noting that it excludes the ensuing system test effort.

Fault tolerance	Code	Rank	Shift	Strip size	Stripe size (data only)	Storage efficiency
1	RAID5	8	-1	24 KB	168 KB	0.875
2	EVENODD2	7	-2	24 KB	168 KB	0.778
2	RDP	8	-2	24 KB	144 KB	0.750
2	X–Code	7	0	20 KB	140 KB	0.714
3	EVENODD3	8	-3	24 KB	168 KB	0.700
3	WEAVER3	8	-	24 KB	96 KB	0.500

Table 1: RAID codes and their layouts used in our evaluations. These layouts vary in physical arrangement, efficiency and performance. EVENODD2 and EVENODD3 are the EVENODD codes for 2 and 3 fault-tolerance respectively. WEAVER3 is the WEAVER code for 3 fault-tolerance. For fairness, we chose layout setting such that the strip size and rank remained relatively same across RAID codes.

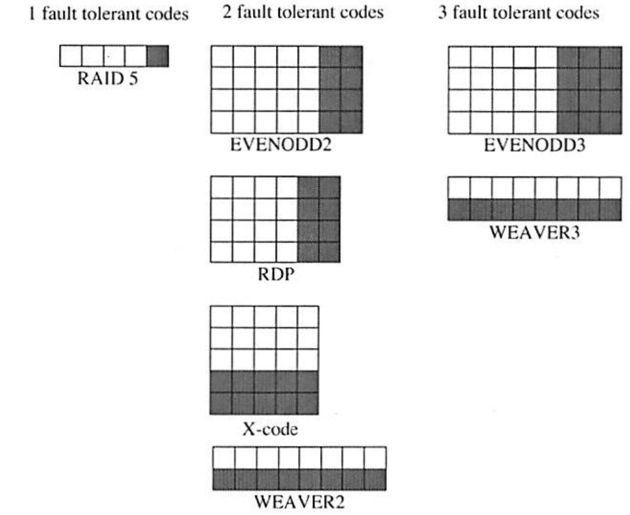


Figure 8: A visual guide to RAID codes studied. Each square corresponds to an element on a disk. Columns of elements are physically layed contiguously on a disk. White squares represent data elements while shaded squares represent parity elements.

Parameter	Setting
Cache size	128 MB
Page size	4 KB
Memory bus bandwidth	1 GBps
I/O bus bandwidth	500 MBps
Disk capacity	18 GB
Disk interconnect BW	150 MBps
Speed	7200 RPM
Single track seek	1.086 ms
Full seek	12.742 ms
Replacement policy	LRU
Window size (W)	0 or 2
Total write ops	100000

Table 2: Parameters used in evaluating the efficacy of Optimizer. Values were chosen to reflect a modest RAID controller. Window size was zero for BASELINE and

8.2 Efficacy of Optimizer

Although not central to the value proposition of REO, we have attempted to quantify the benefits of the Optimizer within the RAID Engine. For this we built a simulation model that included memory and I/O buses and integrated it into `disksim` [13], a disk simulator with fairly accurate disk and array models. We simulated the setup shown in Figure 1.

8.2.1 Setup

Table 2 lists the fixed parameters for our experiment and their values. We chose parameters corresponding to a modest RAID adapter [22].

To realistically quantify the value of the Optimizer we chose trace workloads summarized in Table 3. DS1, P5, P13 and P14 are described by Hsu [20] while TP1, TP2 and SPC1 are publicly available [39]. Given varying durations and intensities of these traces we ran our experiments for a fixed number (100,000) of write I/Os. We did two transformations on the raw traces. First we time shifted them to begin at $t = 0$. Second, we folded multiple LUNs in each trace into a single LUN using appropriate block offsets.

The traces in Table 3 all have a fair amount of random I/O in them. This was a deliberate choice (over picking predominantly sequential workloads) in order to make it more challenging for the Optimizer.

While running a trace, the simulator generated each I/O at the (relative) time specified in the trace. After 100,000 writes were generated, the workload was stopped and the remaining dirty pages in the cache were flushed. The run was deemed complete when there were no more dirty pages left. At the end of the run we extracted total access time from each disk in the volume. Access time for a disk request is the sum of the positioning time (includes seek, rotation, head switching, and settling times) and transfer time. The total access time is computed by summing the access times for all disk

Name	Description	LUNs	Total Size(GB)	Write(%)	Duration
DS1	SAP workload	13	46.91	90	35 mins
TP1	Transaction processing	24	26.01	60	48 mins
TP2	Transaction processing	19	8.25	17	1 hr
P5	Workstation file system workload	2	5.54	70	4 hr
P13	Workstation file system workload	3	3.24	57	26 hr
P14	Workstation file system workload	3	6.04	71	10 hr
SPC1	Storage performance council benchmark [39]	1	7.50	60	56 mins

Table 3: Traces used in this study. We show statistics for the first 100000 write I/Os of each trace that were used in our evaluations.

requests (read and write). For a given workload, total access time is a good measure of the total work done by the disks. During each run we counted the total number of bytes moved on the memory bus. We call this count the memory bus usage.

GRADIENT's objective was set to minimize IOC over XOR since disks tend to be the bottleneck. For faults we modeled static configurations where n disks in a rank were marked as failed for a n failure configuration prior to starting the trace playback. REO performed all I/O assuming such a degraded layout.

To factor out sensitivity to system settings and trace specific patterns, we normalized the total disk access time and the average memory bandwidths for GRADIENT to those for BASELINE. In Table 4 we show these normalized values. Total disk access time and average memory bandwidth are both lower-the-better measures. This implies that a value below 1.0 means that the optimizer using GRADIENT "outperformed" BASELINE.

For brevity, we averaged the results from each of the six application traces (excluding the synthetic benchmark SPC1) for the same layout and fault configuration. This summary from equally weighted summarization of the database and filesystem traces is labeled "Prototypical Workload" in Table 4. The results for SPC1 trace are shown in separate columns.

8.2.2 Discussion of results

Table 4 summarizes the comparison of GRADIENT to the BASELINE which approximates IBM ServeRAID adapters[22]. Since GRADIENT was setup to minimize disk I/Os, we observe that for both workloads, entries in the "Total access time" columns are less than 1.0, a measure of outperformance by GRADIENT. Of interest are the entries in the "Memory bus usage" columns. In some settings e.g., EVENODD3 with 3 faults, the savings in disk access times comes at the expense of increased memory bus usage. This occurs because the heuristic favors a strategy that minimizes IOC over XOR. An example case when this can occur is if, during a write, PI is chosen for an initial set of affected parity pages within a window. Even when it might be cheaper from a memory bandwidth standpoint to choose PC for the subsequent affected parity pages within the window, the heuristic will favor PI in order to minimize IOC. This leads to increased memory bus usage at the expense of reduced disk I/O.

Another reason why GRADIENT outperforms BASELINE on total access times for RAID 5 layouts is because of a bigger window size. Bigger window sizes improve the possibility for fewer and larger sequential I/Os.

On average, there was a modest (4—8%) reduction in disk service times using GRADIENT. The fact that the Optimizer can be competitive w.r.t. other hand tuned RAID implementations is the more important take away rather than the magnitude of its outperformance.

8.3 CPU overhead

Since we used simulations, we could not measure CPU overheads for REO overall or for GRADIENT (over BASELINE). A true measure of CPU overhead is highly sensitive to how a specific implementation is written or the compiler flags used, deployment environment, etc. Such factors are hard to extrapolate to any implementation. However, given the relative speeds of processors and disks, and the modest overhead a costing routine imposes, we believe it to be minor compared to the reduced disk service time.

9 Adaptations

In this section we discuss adaptations of REO to future trends in I/O architecture.

9.1 XOR architectures

Traditional RAID controllers have included hardware support for XOR. An XOR engine, typically built into the memory controller, allows the embedded processor to offload XOR calculations [24]. The approach for computing XOR cost metric discussed in Section 5 reflects the presence of an XOR engine. One recent trend to reduce this cost is to use commodity processors to do XOR as well as I/O handling. This allows leveraging L2 data caches in these processors by combining multiple

Code	Disk failures	Prototypical workload		SPC1 workload	
		Total Access Time	Memory bus usage	Total access time	Memory bus usage
RAID5	0	0.93	0.99	0.90	1.00
	1	0.97	0.99	0.93	0.99
EVENODD2	0	0.97	0.99	0.93	0.90
	1	0.98	0.96	0.94	0.98
	2	0.99	1.08	0.92	1.17
RDP	0	0.98	0.99	0.97	0.93
	1	0.97	0.94	0.95	1.00
	2	0.98	1.04	0.93	1.25
X–Code	0	0.99	1.01	0.96	0.98
	1	0.98	0.85	0.97	0.90
	2	0.98	0.88	0.95	0.97
EVENODD3	0	0.97	0.91	0.92	0.82
	1	0.96	0.94	0.92	0.98
	2	0.97	1.07	0.90	1.19
	3	0.97	1.10	0.88	1.31
WEAVER3	0	0.97	0.99	0.96	0.98
	1	0.93	0.83	0.99	0.90
	2	0.97	0.84	1.00	0.92
	3	0.98	0.86	1.00	0.97

Table 4: Results from trace experiments. All entries in the table are normalized measures for GRADIENT w.r.t. BASELINE. An entry lower than 1.0 implies that the optimizer using GRADIENT "outperformed" BASELINE for that particular setting. Entries under "Prototypical workload" were obtained by averaging individual results for the three database and three filesystem traces. Disk failures were were assumed to be known at start of the run. Notice that, under some layouts and fault configuration, GRADIENT minimizes total access times as the expense of memory bus usage. This is consistent with its resource optimization objective of minimizing disk I/O. REO with GRADIENT is able to modestly reduce disk I/O at the expense of increased memory bus usage for common workloads.

memory fetches for a set of XOR operations with overlapping inputs into a single fetch (for each) of operands. Chunks of operand pages are fetched into the L2 cache and the resultant pages are stored into the L2 cache.

Adapting REO to this XOR architecture is simple. First, a change must be made to the algorithm that costs an IO plan for XOR. From Section 5.2, while computing the total XOR cost for an IO plan, one can count the number of non-zero entries in X in lieu of summing up all its entries. This costing change reflects the memory bandwith used when the processor calculates XOR. The second change is to the execution engine that computes XORs. If the CPU were to do the XOR operations in the I/O plan in Section 4.5 then, using the costing scheme in the previous paragraph XOR = 10.

In our experience with REO, this change of XOR calculation eliminated the memory bandwidth penalty EVENODD incurred over RDP reported by Corbett et al. [12] since common sub-expressions get automatically eliminated. Such optimizations are possible for deployments that use XOR engines at the expense of additional CPU overhead.

9.2 Hierarchical RAID architectures

Hierarchical RAID schemes are structured in layers where one RAID code is used at the top level and another at the lower level. Such layering can boost fault-tolerance at the expense of storage efficiency. For sometime now, commercial RAID controllers have supported RAID 51, a hierarchical scheme which layers a RAID 5 layout over a RAID 1 layout. Of greater relevance to new RAID systems are novel intra-disk redundancy schemes like those proposed by Dholakia et. al. [10]. The goal of their scheme (called SPIDRE) is to build intra-disk redundancy aimed at reducing hard error rates in presence of correlated failures. In their analysis they show that hierarchical RAID schemes that used EVENODD over disks that internally used SPIDRE had 1000x better data reliability over plain EVENODD over the same disks for common correlated sector errors.

REO can be adapted to handle such RAID scheme. To do so, one must the tensor for the product code. Working off this tensor, REO can generate I/O plans for the hierarchical RAID scheme. Any hierarchical RAID scheme will increase the number of affected parity pages for a read or write emphasizing the need for an efficient heuristic for strategy selection.

9.3 Distributed RAID architectures

Another trend in storage architectures is the rise of clustered storage systems [36, 9]. In these systems, data is striped across nodes, each of which has a network connection, processor, memory and a bunch of disks. Layouts span nodes instead of disks in traditional RAID systems. Such architectures can allow for scaling of capacity, reliability and performance at low cost [34].

Adapting to such distributed architectures requires changes to the Execution Engine. Access and updates to data striped using distributed RAID must include some serialization and recovery protocol to handle (a) transient errors from the network and/or nodes, (b) access to data from multiple clients [1], and (c) untrusted nodes [16]. Any of these proposed schemes can be implemented within the Execution Engine without changes to the RAID Engine.

10 Conclusions

We have shown REO to be an ideal solution to the problem of providing a variety of RAID codes without increasing firmware complexity. To our knowledge, REO is unique in its ability to be simultaneously flexible (supporting any XOR-based RAID code), simple (unifying fault-free and fault-ridden code paths), and self-tuning. Not only is it competitive relative to existing RAID implementations, but provides modest performance improvements for a wide range of workloads.

One possible future work would be to leverage REO for adapting data layout based on reliability, performance and efficiency attributes [41, 5].

Acknowledgement

The authors would like to thank Richard Golding, KK Rao, and Veera Dheenadayalan for feedback; Windsor Hsu and Binny Gill for providing traces; Jiri Schindler and anonymous reviewers for comments that helped improve this paper.

References

[1] K. Amiri, G. Gibson, and R. Golding. Highly concurrent shared storage. In *20th Intl. Conf. on Distributed Computing Sys.*, April 2000.

[2] Archivas. http://www.archivas.com/.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD:An optimal scheme for tolerating double disk failure in RAID architectures. *IEEE Transactions on Computers*, 44(2):192–202, 1995.

[4] M. Blaum, J. Bruck, and A. Vardy. MDS array codes with independent parity symbols. *IEEE Transactions on Information Theory*, 42(2):529–542, March 1996.

[5] E. Borowsky, R. Golding, A. Merchant, L. Schrier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS, 1997.

[6] Aarohi Communications. http://www.aarohi.net.

[7] V. Deenadhayalan, J. Hafner, K. Rao, and J. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies*, pages 183–196, San Francisco, CA USA, December 2005.

[8] Intel Digital Enterprise Group Storage Components Division. SCD roadmap update. Powerpoint slide deck, March 2005.

[9] M. Abd el-Malek et. al. Ursa Minor: Versatile cluster-based storage. In *USENIX File and Storage Technologies*, pages 59–72, 2005.

[10] A. Dholakia et. al. Analysis of a new intra-disk redundancy scheme for high-reliability raid storage systems in the presence of unrecoverable errors. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 373–374, 2006.

[11] C. Fleiner et. al. Reliability of modular mesh-connected intelligent storage brick systems. *IBM Journal of Research and Development*, 50(2–3), 2006.

[12] P. Corbett et. al. Row-diagonal parity for double disk failure. In *Proceedings of the Third USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.

[13] G. Ganger, B. Worthington, and Y. Patt. The DiskSim Simulation Environment - Version 2.0 Reference Manual.

[14] G. Gibson, W. Courtright, M. Holland, and J. Zelenka. RAIDframe: Rapid prototyping for disk arrays. Technical Report CMU-CS-95-200, CMU, 1995.

[15] G. Gibson and D. Patterson. Designing disk arrays for high data reliability. *Journal of Parallel and Distributed Computing*, 17(1–2):4–27, 1993.

[16] G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems & Networks*. IEEE, June 2004.

[17] J. Gray. Storage bricks. FAST keynote, also as http://www.research.microsoft.com/Gray/talks/Gray_Storage_FAST.ppt, January 2002.

[18] J. Gray and C. van Ingen. Empirical measurements of disk failure rates and error rates. Technical Report MSR-TR-2005-166, Microsoft Research, December 2005.

[19] J. Hafner. WEAVER codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies*, pages 211–224, San Francisco, CA USA, December 2005.

[20] W. Hsu and A. Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2), 2003.

[21] C. Huang and L. Xu. STAR: An efficient coding scheme for correcting triple storage node failures. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies*, San Francisco, CA USA, 2005.

[22] Adaptec Inc. Adaptec SCSI RAID 2130SLP. http://www.adaptec.com/en-US/products/raid/ultra320_pcix/ASR-2130S/index.htm.

[23] Network Appliance Inc. Netapp primary storage. http://www.netapp.com/products/filer/.

[24] Intel. Intel IOP I/O Processor Chipset. http://www.intel.com/design/iio/iop331.htm.

[25] iVivity. http://www.ivivity.com/.

[26] Lawrence Livermore National Laboratory. Fifth Generation ASC Platform - Purple. http://www.llnl.gov/asc/platforms/purple/.

[27] Aristos Logic. http://www.aristoslogic.com/.

[28] F. MacWilliams and N. Sloane. *The Theory of Error-correcting Codes*. North Holland, 1997.

[29] J. Menon and D. Mattson. Comparison of sparing alternatives for disk arrays. In *Proc. International Symposium on Computer Architecture*, pages 318–329, May 1992.

[30] R. Muntz and J. Lui. Performance analysis of disk arrays under failure. In *Proceedings of the 16th VLDB Conference*, pages 162–173, 1990.

[31] LeftHand Networks. http://www.lefthandnetworks.com/.

[32] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings ACM SIGMOD*, pages 109–116. ACM, June 1988.

[33] J. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software Practice and Experience*, 27(9):995–1012, 1997.

[34] K. Rao, J. Hafner, and R. Golding. Reliability for networked storage nodes. In *International Conference on Dependable Systems and Networks*, 2006.

[35] D. Stodolsky, M. Holland, W. Courtright, and G. Gibson. Parity logging disk arrays. *ACM Transactions on Computer Systems*, 12(3):206–235, 1994.

[36] Isilon Systems. Uncompromising reliability through clustered storage. http://www.isilon.com/media/pdf/Isilon _Uncompromising_Reliability.pdf, May 2005.

[37] Terrascale Technologies. http://www.terrascale.com/.

[38] K. Treiber and J. Menon. Simulation study of cached RAID5 designs. In 1^{st} *IEEE Symposium on High-Performance Computer Architecture*, pages 186–197. IEEE, January 1995.

[39] Various. Storage Performance Council Benchmark Version 1.0. http://www.storageperformance.org.

[40] L. Vepstas. Linux RAID. http://www.tldp.org/HOWTO/Software-RAID-0.4x-HOWTO.html, November 1998.

[41] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst.*, 14(1):108–136, 1996.

[42] L. Xu and J. Bruck. X–code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45:272–276, 1999.

PRO: A Popularity-based Multi-threaded Reconstruction Optimization for RAID-Structured Storage Systems

Lei Tian[†‡], Dan Feng[†‡], Hong Jiang[§], Ke Zhou[†‡],
Lingfang Zeng[†‡], Jianxi Chen[†‡], Zhikun Wang[†‡], Zhenlei Song[†]
[†]*Huazhong University of Science and Technology*
[‡]*Wuhan National Laboratory for Optoelectronics*
[§]*University of Nebraska-Lincoln*
E-mail:{dfeng, k.zhou, ltian}@hust.edu.cn, jiang@cse.unl.edu

Abstract

This paper proposes and evaluates a novel dynamic data reconstruction optimization algorithm, called *p*opularity-based multi-threaded *r*econstruction *o*ptimization (PRO), which allows the reconstruction process in a RAID-structured storage system to rebuild the frequently accessed areas prior to rebuilding infrequently accessed areas to exploit access locality. This approach has the salient advantage of simultaneously decreasing reconstruction time and alleviating user and system performance degradation. It can also be easily adopted in various conventional reconstruction approaches. In particular, we optimize the disk-oriented reconstruction (DOR) approach with PRO. The PRO-powered DOR is shown to induce a much earlier onset of response-time improvement and sustain a longer time span of such improvement than the original DOR. Our benchmark studies on read-only web workloads have shown that the PRO-powered DOR algorithm consistently outperforms the original DOR algorithm in the failure-recovery process in terms of user response time, with a 3.6%~23.9% performance improvement and up to 44.7% reconstruction time improvement simultaneously.

1. Introduction

Reliability and availability are the most common issues that administrators of storage systems are concerned with and directly affect the end users of such systems. Frequent or long downtimes and data losses are clearly intolerable to the end users. Research has shown that 50 percent of companies losing critical systems for more than 10 days never recover, 43 percent of companies experiencing a disaster never reopen, and 29 percent of the remaining close within two years [1]. For some companies that do survive long-term performance degradation, the resulting penalties can become too costly to ignore. For example, for data services at a large-scale data center, a typical Service Level Agreement specifies the percentage of transaction response times that can exceed a threshold (e.g. seconds), and the penalties for failing to comply can be very costly [2].

As a result, redundant storage systems, such as RAID [3], have been widely deployed to prevent catastrophic data losses. Various RAID schemes employ mirroring, parity-encoding, hot spare and other mechanisms [4] to tolerate the failures of disk drives. If a disk failure occurs, RAID will automatically switch to a recovery mode, often at a degraded performance level, and activate a background reconstruction process for data recovery. The reconstruction process will rebuild data units of the failed disk onto the replacement disk while RAID continues to serve I/O requests from clients. After all of the contents are rebuilt, RAID returns the system to the normal operating mode.

Advances in storage technology have significantly reduced cost and improved performance and capacity of storage devices, making it possible for system designers to build extremely large storage systems comprised of tens of thousands or more disks with high I/O performance and data capacity. However, reliability of individual disk drives has improved very slowly, compared to the improvement in their capacity and cost, resulting in a very high overall failure rate in a system with tens of thousands of disk drives. To make matters worse, the time to rebuild a single disk has lengthened as increases in disk capacity far outpace increases in disk bandwidth, lengthening the "window of vulnerability" during which a subsequent disk failure (or a series of subsequent disk failures) causes data loss [5]. Furthermore, many potential applications of data replication or migration, such as on-line backup and capacity management, are faced with similar challenges as the reconstruction process.

Therefore, the efficiency of the reconstruction algorithm affects the reliability and performance of RAID-structured storage systems directly and significantly. The goals of reconstruction algorithms are (1) to minimize the reconstruction time to ensure system reliability and (2) to minimize the performance degradation to ensure user and system performance, simultaneously. Existing effective approaches, such as disk-oriented reconstruction (DOR) [6] and pipelined reconstruction (PR) [7], optimize the recovery workflow to improve the reconstruction process's parallelism. Based on stripe-oriented reconstruction (SOR) [8], both DOR and PR execute a set of reconstruction processes to rebuild data units by exploiting parallelism in processes and pipelining. While they both exploit reconstruction parallelism, the problem of optimizing reconstruction sequence remains unsolved. In a typical recovery mode, the reconstruction process rebuilds data units while RAID continues to serve I/O requests from the clients, where clients' accesses can adversely and severely affect the reconstruction efficiency because serving clients' I/O requests and reconstruction I/O requests simultaneously leads to frequent long seeks to and from the multiple separate data areas. Optimizing the reconstruction sequence with users' workload characteristics, we believe, is a key in improving existing and widely-used reconstruction approaches such as those mentioned above.

In this paper, we propose a Popularity-based multi-threaded Reconstruction Optimization (PRO) algorithm to optimize the existing reconstruction approaches, which exploits I/O workload characteristics to guide the reconstruction process. The main idea behind our PRO algorithm is to reconstruct high-popularity data units of a failed disk, which are the most frequently accessed units in terms of the workload characteristics, prior to reconstructing other units. Furthermore, by fully exploring the access patterns, the PRO algorithm has the potential to recover many units ahead of users' accesses with high probability to avoid performance degradation caused by recovery.

To the best of our knowledge, little research effort has been directed towards integrating workload characteristics into the reconstruction algorithm. Most of the existing reconstruction algorithms perform a sequential reconstruction on the failed disk regardless of the workload characteristics. Taking into account the workload distribution (such as spatial locality, temporal locality, etc.), the PRO approach attains a flexible and pertinent reconstruction strategy that attempts to reduce reconstruction time while alleviating the performance degradation. PRO achieves this by inducing a much earlier onset of response-time performance improvement and sustaining a longer time span of such improvement while reducing reconstruction time during recovery. We implement the PRO algorithm based on DOR, and our benchmark studies on read-only web workloads have shown that the PRO algorithm outperforms DOR by 3.6%~ 23.9% in user response time and by up to 44.7% in reconstruction time, simultaneously. Because the PRO algorithm is specifically designed to only generate the optimal reconstruction sequence based on workload characteristics without any modification to the reconstruction workflow or data layout, it can be easily incorporated into most existing reconstruction approaches of RAID and achieve significant improvement on system reliability and response time performance.

The rest of this paper is organized as follows. Related work and motivation are presented in Section 2. In Section 3, we describe the algorithm and implementation of the PRO approach. Performance evaluation and discussions are presented in Section 4. We conclude the paper in section 5 by summarizing the main contributions of this paper and pointing out directions for future research.

2. Related Work and Motivation

In this section, we first present the background and related work on the reconstruction issues. Second, we reveal the ubiquitous properties of popularity and locality of typical workloads and thus elucidate the motivations for our research.

2.1. Existing Reconstruction Approaches

There has been substantial research reported in the literature on reliability and recovery mechanisms in RAID or RAID-structured storage systems. Since the advent of disk arrays and RAID, researchers have been working to improve reliability and availability of such systems by devising efficient reconstruction algorithms for the recovery process, with several notable and effective outcomes.

There are two general approaches to disk array reconstruction depending on the source of improvement. The first general approach improves performance by reorganizing the data layout of spare or parity data units during recovery.

Hou et al. [9] considered an approach to improving reconstruction time, called distributed sparing. Instead of using a dedicated spare disk which is unused in the normal mode, they distribute the spare space evenly across the disk array. Their approach can result in improved response times and reconstruction times because one extra disk is available for normal use and less data needs to be reconstructed on a failure. As an extension to the distributed sparing approach in large-scale distributed storage systems, Xin et al. [10] presented a fast recovery mechanism (FARM) to dramatically lower the probability of data loss in large-scale storage systems.

The second general approach improves performance by optimizing the reconstruction workflow. Because this approach needs not modify the data organization of RAID, it is more prevalent in RAID implementations. Therefore, it is also the approach that our proposed algorithm will be based on.

Disk-oriented reconstruction (DOR) and pipelined reconstruction (PR) are two representative and widely-used reconstruction approaches. The DOR algorithm was proposed by Holland [11] to address the deficiencies of both the single-stripe and parallel-stripe reconstruction algorithms (SOR). Instead of creating a set of reconstruction processes associated with stripes, the array controller creates a number of processes, with each associated with one disk. The advantage of this approach is that it is able to maintain one low-priority request in the queue for each disk at all times, thus being able to absorb all of the array's bandwidth that is not absorbed by the users. Although DOR outperforms SOR in reconstruction time, the improvement in reliability comes at the expense of performance in user response times.

To address the reliability issue of continuous-media servers, Lee and Lui [7] presented a track-based reconstruction algorithm to rebuild lost data in tracks. In addition, they presented a pipelined reconstruction (PR) algorithm to reduce the extra buffers required by the track-based reconstruction algorithm. Muntz and Lui [12] conducted a performance study on reconstruction algorithms using an analytical model. Their first enhancement, reconstruction with redirection of reads, uses the spare disk to service disk requests to the failed disk if the requested data has already been rebuilt to the spare disk. They also proposed to piggyback rebuild requests on a normal workload. If a data block on the failed disk is accessed and has not yet been rebuilt to the spare disk, the data block is regenerated by reading the corresponding surviving disks. It is then a simple matter of also writing this data block to the spare disk.

The basic head-following algorithm [11] attempts to minimize head positioning time by reconstructing data and parity in the region of the array currently being accessed by the users. The problem with this approach is that it leads to almost immediate deadlock of the reconstruction process. Since the workload causes the disk heads to be uncorrelated with respect to each other, head following causes each reconstruction process to fetch a reconstruction unit from a different parity stripe. The buffers are filled with data units from different parity stripes and hard to be freed, and so reconstruction deadlocks.

Assuming that the probability of a second disk failure (while the first failed disk is under repair) is very low, Kari et al. [13] presented a delayed repair method to satisfy the response time requirement, which introduced a short delay between repair requests to limit the number of repair read (or write) requests that any user disk request might need to wait. With regard to RAID-structured storage systems, it may not be practical to adopt such a delayed repair method directly because of the relatively high probability of a second disk failure.

To address the problem of performance degradation during recovery, Vin et al. [14] integrated the recovery process with the decompression of video streams, thereby distributing the reconstruction process across the clients to utilize the inherent redundancy in video streams. From the viewpoint of a file system's semantic knowledge, Sivathanu et al. [15] proposed a live-block recovery approach in their D-GRAID, which changes the recovery process to first recover those blocks which are live (i.e., belong to allocated files or directories).

Of the above related work on disk array reconstruction or recovery based on reconstruction workflow optimizations, the DOR algorithm still dominates in its applications and implementations; whereas, the others can be arguably considered either rooted at DOR or DOR's extensions or variations. In general, while DOR absorbs the disk array's bandwidth, its variations attempt to take advantage of user accesses. On the other hand, our proposed PRO algorithm improves over these approaches by not only preserving the inherent sequentiality of the reconstruction process, but also dynamically scheduling multiple reconstruction points and considering the reconstruction priority. What's more important, the PRO algorithm makes use of the access locality in I/O workloads, which is ignored by the above reconstruction algorithms, and improves the response-time performance and the reliability performance simultaneously.

2.2. Popularity and Locality of Workloads

A good understanding of the I/O workload characteristics can provide a useful insight into the reason behind DOR's inability to minimize application's performance degradation, thus helping us improve the DOR algorithm and other related algorithms to address this problem. In particular, workload characteristics such as temporal locality and spatial locality can be exploited to alleviate performance degradation while reducing reconstruction time. Experience with traditional computer system workloads has shown the importance of temporal locality to computer design [16].

In many application environments, 80% accesses are always directed to 20% of the data, a phenomenon that has long been known as Pareto's Principle or "The 80/20 Rule" [17]. Pareto's Principle points to the existence of temporal locality and spatial locality in various kinds of I/O workloads. The presence of access locality in I/O workloads has been well known in the literature. Gomez & Santonja [17] studied three sets of I/O traces provided by Hewlett-Packard Labs, and showed that some of the blocks are extremely hot and popular while other blocks are rarely or never accessed. In the media workload, Cherkasova and Gupta [18] found that 14%~30% of the files accessed on the server accounted for 90% of the media sessions and 92%~94% of the bytes transferred, and were viewed by 96%~97% of the unique clients.

Studies have further indicated that access locality is more pronounced in workload characteristics of web servers [19], and identified three distinct types of web access locality, namely, static, spatial and temporal locality. Static locality refers to the observation that 10% of files accessed on a web server typically account for 90% of the server requests and 90% of the bytes transferred [20]. Roselli et al. [21] found that for all workloads, file access patterns are bimodal in that most files tend to be mostly-read or mostly-written. The web workload has far more read bandwidth than any other workload but has relatively little write bandwidth. Sikalinda et al. [22] found that in the web search workload almost all the operations are reads (i.e., 99.98% of the total number of operations). Since typical on-line transaction processing type (OLTP-like) workloads are read-dominated [13], the load on the surviving disks increased by typically between 50-100% in the presence of a disk failure. This severely degrades the performance as observed by the users, and dramatically lengthens the period of time required to recover the lost data and store it on a replacement disk.

Motivated by insightful observations made by other researchers and by our own research, we propose to integrate temporal locality and spatial locality into the reconstruction algorithms to improve their effectiveness. If the frequently accessed data units could be reconstructed prior to reconstructing all others, the effect of performance degradation can be potentially hidden from the users in their subsequent accesses to the same data units, especially during the recovery process.

3. Design and Implementation of PRO

Due to the aforementioned shortcomings of existing parallel reconstruction approaches (such as DOR and PR), a solution must be sought to optimize the reconstruction sequence. Based on prior research by other researchers and by us, we believe that the key to our solution is to exploit the workload characteristics into the reconstruction algorithm. By making an appropriate connection between the external workload and the internal reconstruction I/O activities, we propose a Popularity-based multi-threaded Reconstruction Optimization algorithm (PRO) to combine the locality of the workloads with the sequentiality of the reconstruction process.

The main idea behind the PRO algorithm is to monitor and keep track of the dynamic popularity changes of data areas, thus directing the reconstruction process to rebuild highly popular data units of a failed disk, prior to rebuilding other units. More specifically, PRO divides data units on the replacement disk into multiple non-overlapping but consecutive data areas, called "hot zones". Correspondingly, PRO employs multiple independent reconstruction threads, where each reconstruction thread is responsible for one of the hot zones and the priority of each thread is determined by the latest frequency of users' accesses to its hot zone. The PRO algorithm adopts a priority-based time-sharing scheduling algorithm to schedule the reconstruction threads. The scheduler activates the thread with the highest priority periodically by allocating a time slice to the thread that begins rebuilding the remaining data units of its hot zones until the time slice is used up.

Different from the strictly sequential sequence of most existing reconstruction approaches, PRO generates a workload-following reconstruction sequence to exploit the locality of workload characteristics.

In this section, we first outline the design principles behind PRO, which is followed by a detailed description of the PRO algorithm via an example, as well as an overview of PRO's implementation.

3.1. Design Principles

As Holland concluded [11], a reconstruction algorithm must preserve the inherent sequentiality of the

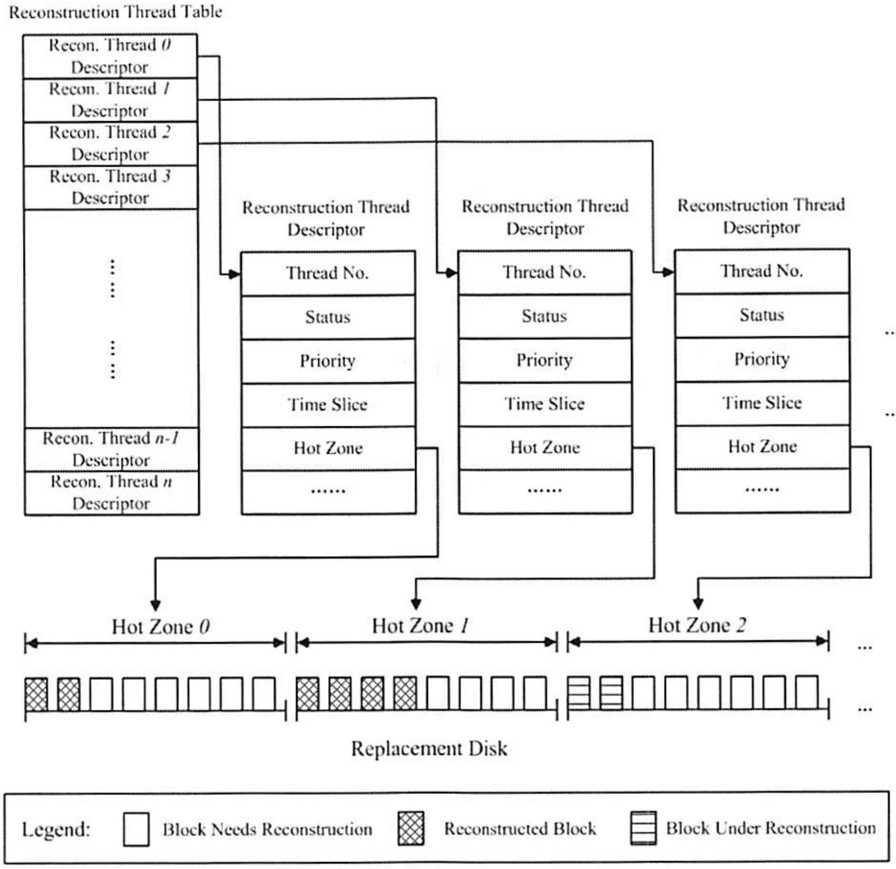

Reconstruction Thread Table
Recon. Thread 0 Descriptor
Recon. Thread 1 Descriptor
Recon. Thread 2 Descriptor
Recon. Thread 3 Descriptor
Recon. Thread n-1 Descriptor
Recon. Thread n Descriptor
Reconstruction Thread Descriptor
Thread No.
Status
Priority
Time Slice
Hot Zone
......
Hot Zone 0
Hot Zone 1
Hot Zone 2
Replacement Disk
Legend:
Block Needs Reconstruction
Reconstructed Block
Block Under Reconstruction

Figure 1: Reconstruction thread descriptor.

reconstruction process, since a disk drive is able to service sequential accesses at many times the bandwidth of random accesses. This leads to the development of the disk-oriented reconstruction (DOR) algorithm and the pipelined reconstruction (PR) algorithm, and to the rejection of the head-following approaches. On the other hand, to take full advantage of locality of access in the I/O workload, we believe that a reconstruction algorithm should rebuild data units with high-popularity prior to rebuilding low-popularity or no-popularity data units on the failed disk. Consequently, frequent long seeks to and from the multiple separate popular data areas result in seek and rotation penalties for multiple reconstruction points.

To strike a good balance between the above two seemingly conflicting goals of maintaining sequentiality and exploiting access locality, the PRO algorithm effectively combines "global decentralization" with "local sequentiality" in the reconstruction process. Inspired by the design principles of classical time-sharing operating systems, PRO adopts the ideas of "divide and conquer" and "time-sharing scheduling" to achieve the above goals.

3.2. The Reconstruction Algorithm

To obtain an optimal reconstruction sequence, PRO tracks the popularity changes of data areas and generates a workload-following reconstruction sequence to combine locality with sequentiality.

A description of the PRO algorithm is given as follows.

First, PRO divides data units on the replacement disk into multiple non-overlapping but consecutive data areas called *"hot zones"* (the algorithm for defining the appropriate number of hot zones is described in Section 3.4), with each being associated with a popularity measure determined by the latest frequency of users' accesses to it. Correspondingly, the entire reconstruction process is divided into multiple independent reconstruction threads with each being responsible for rebuilding one of the hot zones. The priority of a reconstruction thread is dynamically adjusted according to the current popularity of its hot zone. Similar to a real thread in operating systems, each reconstruction thread has its reconstruction thread descriptor (see Figure 1

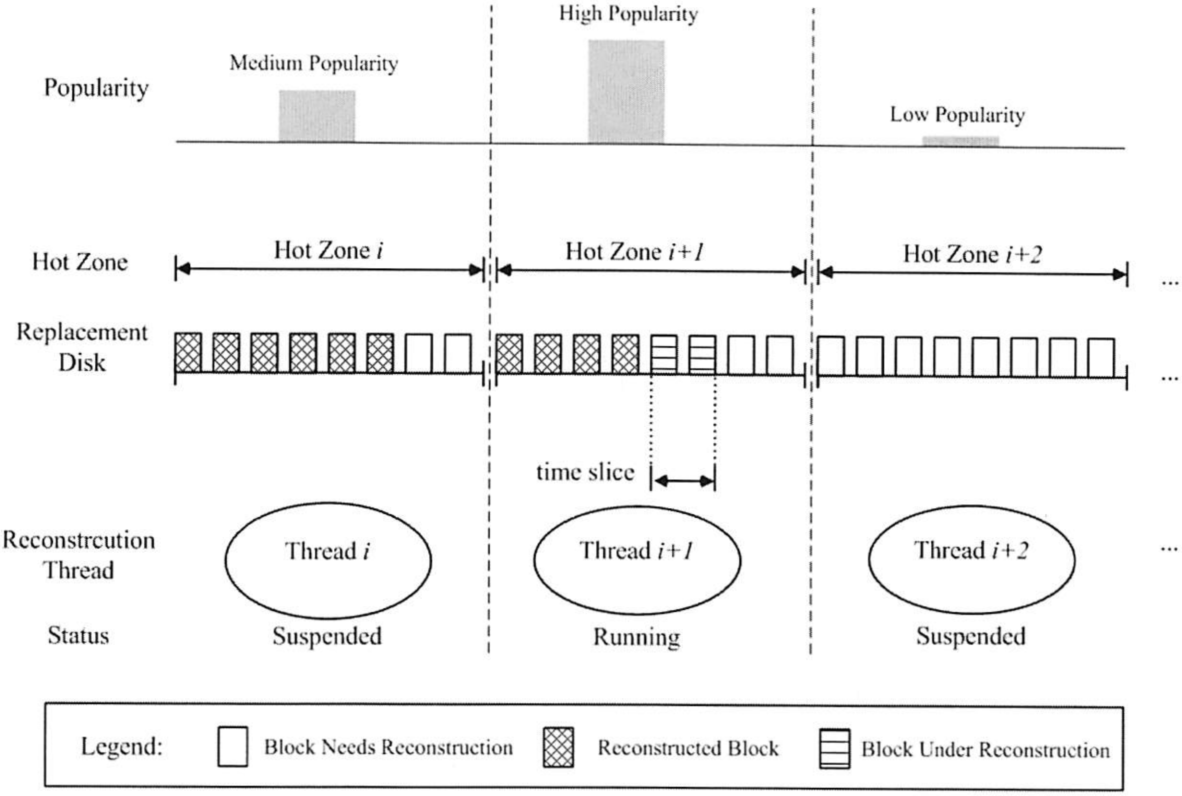


Figure 2: A snapshot of the PRO algorithm at work.

that shows its data structure and content) that includes properties such as status, priority, time slice, etc.

Second, once all of the reconstruction threads are initialized, a reconstruction scheduler selects the reconstruction thread with the highest priority and allocates a time slice to it, which activates this thread to rebuild the remaining data units of its hot zone until the time slice is used up. If the thread runs out of its time slice, the reconstruction scheduler suspends it, re-selects the reconstruction thread that has the current highest priority, and allocates one time slice to it. This process repeats until all the data units have been rebuilt. Figure 2 is a snapshot of the PRO algorithm at work. From Figure 2, one can see three reconstruction threads and their respective hot zones with different popularity. Because the hot zone of the middle thread has the highest popularity, the thread is switched to the running state and allocated a time slice by the reconstruction scheduler. At the same time, the other two threads are suspended due to their low popularity.

The PRO approach uses the reconstruction threads to track the popularity changes of the hot zones and always picks up the most popular data area to rebuild prior to rebuilding others, thus achieving the goal of fully exploiting access locality based on popularity (via global decentralization) during the reconstruction process.

A time slice in PRO is always associated with a number of the consecutive data units. In our implementation, a time slice is set to be 64, that is, 64 consecutive data units. With this approach, PRO achieves the goal of maintaining local sequentiality of the reconstruction process.

After the above step, the data units in the reconstruction queue are workload-following and locally sequential.

3.3. The PRO Structure and Procedures

The PRO architecture consists of three key components: Access Monitor (AM), Reconstruction Scheduler (RS) and Reconstruction Executor (RE). AM is responsible for capturing and analyzing clients' access locality and adjusting popularities of the corresponding hot zones. The responsibility of RS is to select data units of the hot zone with the highest popularity, or the most frequently-accessed data units, generate the corresponding tasks and put them into a FIFO reconstruction task queue. The main function of RE is to fetch jobs from the reconstruction task queue and rebuild the corresponding units on the replacement disk. The operations of each of the three PRO components are detailed below.

Access Monitor (AM):

```
Repeat
    1. AM receives the I/O request and determines its type and address information.
    2. If the type is read and the address is located on the failed disk, increase the popularity of the corresponding hot zone (including this address).
Until (the failed disk has been reconstructed)
```

Reconstruction Scheduler (RS):

```
Repeat
    1. Check the time slice of the reconstruction thread whose status is running.
    2. If the thread has no time slice left, select the reconstruction thread with the highest popularity from all reconstruction threads, and allocate a time slice to it. At the same time, reset the popularity of all reconstruction threads to zero.
    3. Set the status of the chosen reconstruction thread to the running state, and set the status of other threads to the suspended state. Meanwhile, reduce the remaining time slice of the chosen reconstruction thread by one.
    4. The chosen reconstruction thread begins rebuilding a remaining data units in its hot zone, generating a corresponding reconstruction job to obtain this reconstruction unit, and queuing it to the tail of the reconstruction job queue.
    5. If all data units in this hot zone are reconstructed, reclaim the chosen reconstruction and its hot zone.
Until (the failed disk has been reconstructed)
```

Reconstruction Executor (RE):

In fact, the functions of RE are the same as those of the DOR algorithm except for the following subtle difference: RE chooses the next job from the queue, not the sequentially next data unit one by one as the DOR algorithm. The disk array controller creates *C* processes, each associated with one disk. Each of the *C-1* processes associated with a surviving disk executes the following loop:

```
Repeat
    1. Fetch the next job from the reconstruction job queue on this disk that is needed for reconstruction.
    2. Issue a low-priority request to read the indicated unit into a buffer.
    3. Wait for the read to complete.
    4. Submit the unit's data to a centralized buffer manager for XOR, or Block the process if the buffer is full.
Until (all necessary units have been read)
```

The process associated with the replacement disk executes the following operations:

```
Repeat
    1. Request sequentially the next full buffer from the buffer manager.
    2. Issue a low-priority write of the buffer to the replacement disk.
    3. Wait for the write to complete.
Until (the failed disk has been reconstructed)
```

3.4. Implementation Issues

How is the popularity data collected, stored, updated?
Whenever the reconstruction process starts, the Access Monitor begins to track and evaluate the statistics of clients' accesses to the failed disk. The popularity data of each hot zone is stored in the "popularity" counter of the hot zone. It can keep better track of the dynamic changes of popular data areas if the popularity data is collected, stored and updated even after the reconstruction process starts. This, however, leads to a 2% to 4% system performance loss (similar to the impact of the I/O trace tools) while the probability of a disk failure is relatively low.

Once a client accesses a hot zone, the value of its popularity counter is increased by one. Each time the reconstruction scheduler finishes a thread scheduling procedure, all the popularity counters of hot zones are reset to zeros to be ready for a new round of monitoring. This implies that short-term, rather than long-term, popularity history is captured by the PRO implementation. This is based on our belief that the former is more important than the latter during recovery because current or recent past popularity change tends to point to newly and rapidly accessed data areas that should be reconstructed more urgently.

How many zones are used?
In PRO, hot zones are fully dynamically created, updated and reused, including the corresponding

Component	Description
CPU	Intel Pentium4 3GHz
Memory	512M DDR SDRAM
SCSI HBA	LSIlogic 22320R
SCSI Disk	Seagate ST3146807LC
OS	NetBSD 2.1
RAID software	RAIDframe [26]

Table 1: The platform for performance evaluation.

reconstruction threads. At the beginning of the recovery process, there is no reconstruction thread or hot zone.

When a client accesses a data unit, say, numbered N, on the failed disk, a thread will be created and initiated, along with its corresponding hot zone of default length L and range N to $N+L$. After initialization, the reconstruction thread switches to the *alive* status and waits for scheduling. If a client accesses a data unit belonging to this hot zone, its popularity will be increased by one. But if a client accesses a data unit not belonging to any of the existing hot zones, a new thread, along with its new hot zone, will be initialized and created in a manner described above. Assuming that the data unit is numbered M, if $M<N$ and $N\text{-}M<L$, the range of the newly created hot zone is set to be M to $N\text{-}M$ to ensure non-overlapping between two adjacent hot zones, else the range is set to be M to $M+L$. In the other words, the default length L is the maximum threshold. When all data units in a hot zone are rebuilt, the hot zone and the corresponding reconstruction thread will be reclaimed for the reuse purpose.

Clearly, the number and length of hot zones are not fixed but dynamically adjusted by the workload. In the implementation, we set the maximum length of a hot zone to 1024 and the maximum number of hot zones (thus of the reconstruction threads) to 128, that is, the PRO algorithm can track the popularity changes of 128 data areas simultaneously.

How are threads scheduled?
The PRO algorithm adopts a priority-based time-sharing scheduling algorithm to schedule multiple reconstruction threads. It must be noted that the PRO algorithm's scheduler bases its selection decision mostly on a thread's priority (or its corresponding zone's popularity).

4. Performance Evaluations

This section presents results of a comprehensive experimental study comparing the proposed PRO-powered DOR algorithm (PRO for short) with the original DOR algorithm. To the best of our knowledge, the DOR algorithm is arguably the most effective among existing reconstruction algorithms in part because it is implemented in many software and hardware RAID products [23-25] and most widely studied in the literature. This performance study analyzes reconstruction performance in terms of user response time and reconstruction time, and algorithm complexity.

4.1. Experimental Setup

We conducted our performance evaluation of the two algorithms above on a platform of server-class hardware and software (see Table 1). The speed of the Seagate ST3146807LC disks is 10000 rpm, its average seek time is 4.7ms, and its capacity is 147GB. We use 2 SCSI channels and each channel attaches 5 disks.

Because the NetBSD platform has no appropriate benchmark to replay the I/O trace, we implemented a block-level replay tool, RAIDmeter, to evaluate performances of the two reconstruction algorithms since most databases run not on a file system but on the raw devices directly. The main function of RAIDmeter is to replay the traces and evaluate the I/O response time, that is, to open the block device, send the designated read/write requests to the device in terms of the time-stamp and request type of each I/O event in the trace, wait for the completion of the I/O requests and gather the corresponding I/O results, such as the response time, throughput and so on. Since the NetBSD 2.1 operating system cannot support read or write operations to files larger than 2GB, it is very inconvenient to benchmark the performance of disk arrays consisting of hard drives with hundreds of GB capacity each. As a result, we had to limit the capacity of every disk to 5GB. In addition, RAIDmeter adopts the approach of randread [27] to overcome the capacity limitation to support I/O accesses to files of tens of GBs. We believe that RAIDmeter is the best block-level trace-replay tool available for the current NetBSD platform.

4.2. Methodology and Workload

We evaluated our design by running trace-driven evaluations over the web I/O traces identified from the Storage Performance Council [28] because the web workloads tend to most pronouncedly reveal the nature of locality. These traces were collected from a system running a web search engine, and they are all read-dominant and with high locality. Because of the absolute read-domination (99.98%) in all of the web-search-engine I/O traces, we filtered out the write requests and feed only read events to our RAIDmeter replay tool.

Trace Name	Number of Requests	Popularity (20% most active area by size handles so much of total requests)	Average Inter-Arrival Time (ms)
Web Trace 1	842,535	69.99%	3.74
Web Trace 2	999,999	70.12%	3.73
Web Trace 3	999,999	69.89%	5.71

Table 2: The characteristics of three web traces.

RAID Level	Number of Disks	Reconstruction Time (second)								
		Web Trace 1			Web Trace 2			Web Trace 3		
		DOR	PRO	improved	DOR	PRO	improved	DOR	PRO	improved
RAID5	3	1123.8	666.5	40.7%	1058.3	585.2	44.7%	452.3	351.6	22.3%
	5	457.4	353.1	22.8%	374.5	304.1	18.8%	242.8	203.9	16.0%
	7	344.0	319.1	7.2%	325.7	278.5	14.5%	238.2	210.8	11.5%
	9	243.5	240.3	1.3%	231.5	215.3	7.0%	192.4	184.5	4.1%
RAID1	2	1208.1	1157.7	4.2%	938.0	870.1	7.3%	424.2	409.5	3.7%

Table 3: A comparison of PRO and DOR reconstruction times as a function of the number of disks.

RAID Level	Number of Disks	Average User Response Time during recovery (millisecond)								
		Web Trace 1			Web Trace 2			Web Trace 3		
		DOR	PRO	improved	DOR	PRO	improved	DOR	PRO	improved
RAID5	3	31.8	24.2	23.9%	28.5	23.9	16.0%	27.4	23.1	15.6%
	5	21.7	19.3	11.1%	21.0	18.7	11.0%	20.0	17.8	11.3%
	7	25.0	23.8	5.1%	22.5	21.4	4.5%	22.6	20.0	11.8%
	9	19.1	18.2	4.5%	19.6	17.3	11.5%	19.5	18.8	3.6%
RAID1	2	29.5	28.2	11.1%	21.4	20.6	11.0%	18.8	17.8	11.3%

Table 4: A comparison of PRO and DOR user response times as a function of the number of disks.

Owing to the relatively short reconstruction times in our current experimental setup, it may not be necessary to use the full daily-level traces. Thus, we only reserved the beginning part of the traces with lengths appropriate for our current reconstruction experiments. Table 2 shows the relevant information of the web-search-engine I/O traces.

4.3. Reconstruction Performance

We first conducted our performance evaluation of the two reconstruction algorithms on a platform of a RAID-5 disk array consisting of variable number of disks and 1 hot-spare disk, with a stripe unit size of 64KB and a RAID-1 disk array consisting of 2 disks and 1 hot-spare disk, with a stripe unit of 64KB.

Tables 3 and 4 show the measured reconstruction times and user response times of DOR and PRO, respectively, and reveal the efficacy of the PRO algorithm in improving the array's reconstruction time and user response time simultaneously during recovery. Across all given workloads, RAID levels and disk numbers in our experiments, the PRO algorithm almost consistently outperforms the DOR algorithm in reconstruction time and user response time by up to 44.7% and 23.9%, respectively.

It is not uncommon that a second disk drive can fail shortly after a first disk failure in a very large-scale RAID-structured storage system, and thus it is very important to shorten the reconstruction time to avoid a

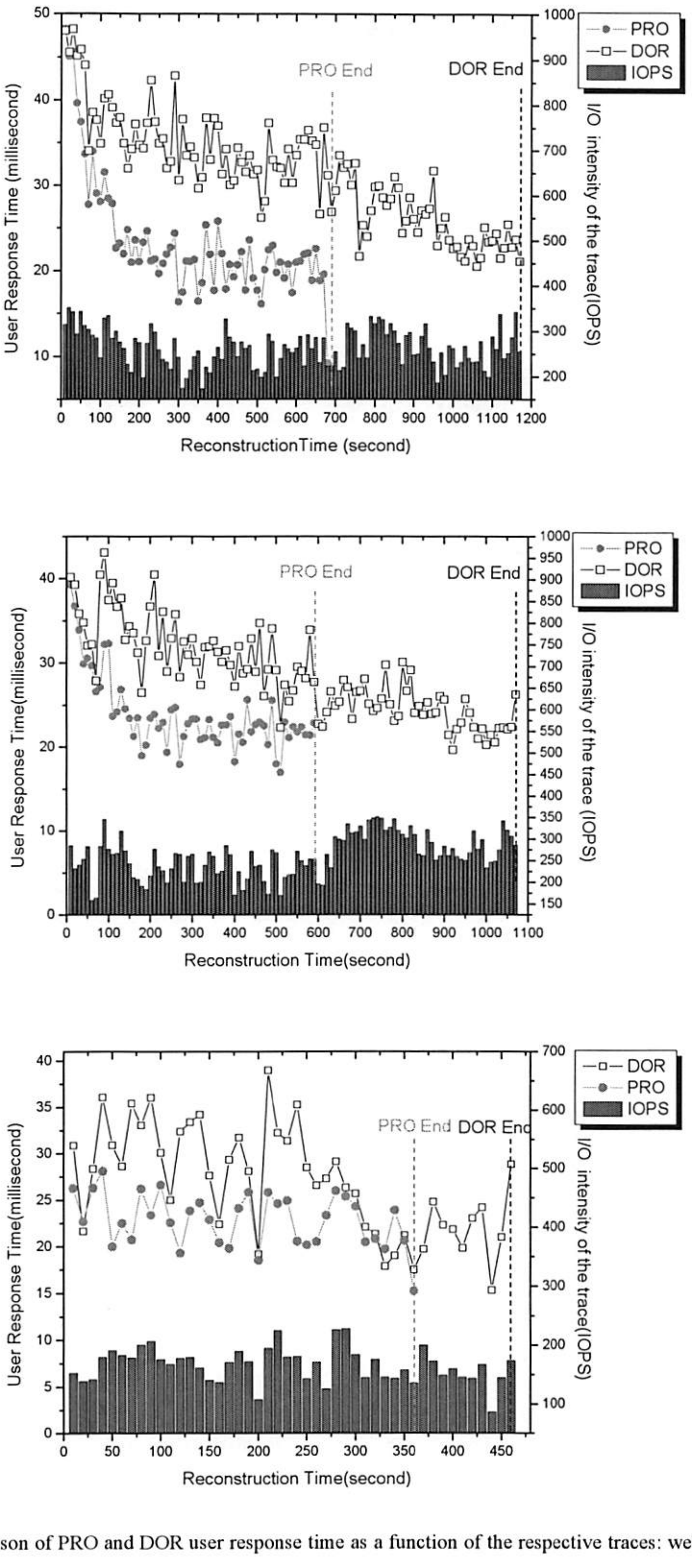


Figure 3 (a), (b), (c): A comparison of PRO and DOR user response time as a function of the respective traces: web trace 1, 2 and 3. In all of the figures, the bottom columns mean the I/O intensity of the trace and the two curves mean the PRO and DOR user response time trend during recovery. The RAID-5 disk array consists of 3 disks and 1 hot-spare disk, with a stripe unit size of 64KB.

In summary, by exploiting access locality the PRO algorithm consistently outperforms the DOR algorithm both in reliability and in performance during recovery.

4.4. Complexity Analysis

Space Complexity

Because the PRO approach is based on the implementation of the DOR algorithm, the memory requirements are almost the same as that for the latter. Compared with DOR, PRO requires extra memory only for the storage of reconstruction thread descriptors. In our actual implementation, each thread descriptor consumes 1KB memory, and if we set the maximum threshold of the number of threads to 128 (as is in our implementation), the extra memory needed for the PRO approach is about 128KB.

Time Complexity

Since the time for each thread-scheduling event is mostly consumed in the selection of the highest-priority thread from all of the candidate threads, we can estimate approximately that the computation overhead of the PRO algorithm is *O(n)*, where n is the total number of the existing threads. If we use a priority queue, the computation complexity can be reduced to *O(logn)*. However, the computation overhead will be negligible on a modern processor compared with the enormous I/O latency of a disk.

Implementation Complexity

We briefly quantify the implementation complexity of PRO. Table 5 lists the lines of code, counted by the number of semicolons and braces, which are modified or added to the RAIDframe. From the table, one can see that very few additions and modifications are needed to add to the reconstruction module. It is clear that most of the complexity is found in the Reconstruction Scheduler, because this module is in fact the bridge between the Access Monitor and the Reconstruction Executor, and its functions are shared with these modules. Compared with the 39691 lines of the RAIDframe implementation, the modifications for PRO occupy only 1.7 % of the total lines.

5. Conclusion

The recovery mechanism of RAID becomes increasingly more critical to the reliability and availability of storage systems. System administrators demand a solution that can reconstruct the entire content of a failed disk as quickly as possible and, at the same time, alleviate performance degradation during recovery as much as possible. However, it is very difficult to achieve these two goals simultaneously.

Component	Mod. Lines	Add. Lines	Total Lines
AM	0	84	84
RS	0	590	590
RE	2	12	14
Total	2	686	688

Table 5: Code complexity for the DOR modifications in the RAIDframe.

In this paper, we developed and evaluated a novel dynamic reconstruction optimization algorithm for redun dant disk arrays, called a Popularity-based multithreaded Reconstruction Optimization algorithm (PRO). The PRO algorithm exploits the access locality of I/O workload characteristics, which is ubiquitous in real workloads, especially in the web server environment. The PRO algorithm allows the reconstruction process to rebuild the frequently-accessed areas prior to building infrequently-accessed areas. Our experimental analysis shows that the PRO algorithm results in a 3.6% ~ 23.9% user performance improvements during recovery over the DOR algorithm. Further, PRO leads to a much earlier onset of performance improvement and longer time span of such improvement than DOR during recovery. More importantly, PRO can reduce the reconstruction time of DOR by up to 44.7%. By effectively exploiting access locality, the PRO algorithm accomplishes the goal of simultaneous improvement of reliability and user and system performance.

However, the PRO algorithm in its current form may not be suitable for all types of workloads, for example, it will not likely work well under write-dominated workload. A good solution for write-dominated workload remains one of our directions for future research on PRO. In addition, current PRO only integrates the access locality into the reconstruction algorithm, without distinguishing or predicting access patterns. We believe that PRO's effectiveness can be increased if the access patterns are discovered, predicted, and incorporated into the reconstruction process, which is another direction of our future research.

Acknowledgments

We would like to thank our shepherd, Andrea C. Arpaci-Dusseau, and the anonymous reviewers for their helpful comments in reviewing this paper. We also thank Greg Oster (the RAIDframe maintainer in NetBSD operating system) at the University of Saskatchewan for his insightful suggestions, and thank Chao Jin, Xiongzi Ge and Qiang Zou for their help in writing this paper.

This work is sponsored by the National Basic Research Program of China (973 Program) under Grant No. 2004CB318201, the Key Basic Research Project under Grant No.2004CCA07400, Project CNGI-04-5-1D, Program for New Century Excellent Talents in University NCET-04-0693, the China NSF under Grant No.60273074, No.60503059, No. 60303032, No. 60603048, and the US NSF under Grant No. CCF-0621526.

References

[1]. D. Wenk. Is 'Good Enough' Storage Good Enough for Compliance? *Disaster Recovery Journal*. 2004.

[2]. Q. Zhu and Y. Zhou. Chameleon: A Self-Adaptive Energy-Efficient Performance-Aware RAID. In *Proceedings of the fourth annual Austin Conference on Energy-Efficient Design(ACEED 2005)*, Poster Session, Austin, Texas, March 1-3, 2005.

[3]. D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the Conference on Management of Data*, 1988.

[4]. G. Gibson, Redundant Disk Arrays: Reliable, Parallel Secondary Storage. MIT Press, 1992.

[5]. Q. Xin, E. L. Miller, T. J. Schwarz, D. D. E. Long, S. A.Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, April 2003.

[6]. M. Holland, G. Gibson, and D. Siewiorek. Fast, On-Line Failure Recovery in Redundant Disk Arrays. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 422-43, 1993.

[7]. Jack Y.B. Lee and John C.S. Lui. Automatic Recovery from Disk Failure in Continuous-Media Servers. IEEE Transaction On Parallel And Distributed Systems, Vol. 13, No. 5, May 2002.

[8]. M. Holland, G.A. Gibson and D. Siewiorek. Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays. Journal of Distributed and Parallel Databases, Vol. 2, No. 3, pages 295-335, July 1994.

[9]. R. Hou, J. Menon, and Y. Patt. Balancing I/O Response Time and Disk Rebuild Time in a RAID5 Disk Array. In *Proceedings of the Hawaii International Conference on Systems Sciences*, pages 70-79, 1993.

[10]. Q. Xin, E. L. Miller and T. J. Schwarz. Evaluation of Distributed Recovery in Large-Scale Storage Systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 172-181, June 2004.

[11]. M. Holland. On-Line Data Reconstruction in Redundant Disk Arrays. Carnegie Mellon Ph.D. Dissertation CMU-CS-94-164, April 1994.

[12]. R. Muntz and J. Lui, Performance Analysis of Disk Arrays Under Failure. In *Proceedings of the 16th Conference on Very Large Data Bases*, 1990.

[13]. H. H. Kari, H. K. Saikkonen, N. Park and F. Lombardi. Analysis of repair algorithms for mirrored-disk systems. IEEE Transactions on Reliability, Vol 46, No. 2, pages 193-200, 1997.

[14]. H.M Vin, P.J. Shenoy and S. Rao. Efficient Failure Recovery in Multi-Disk Multimedia Servers. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 12–21, 1995.

[15]. M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, March 2003.

[16]. A. Smith. Cache Memories. Computing Surveys, Vol 14, No. 3, pages 473–480, 1982

[17]. M. E. Gomez and V. Santonja. Characterizing Temporal Locality in I/O Workload. In *Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'02)*. San Diego, USA, July 2002

[18]. L. Cherkasova and M. Gupta. Analysis of Enterprise Media Server Workloads: Access Patterns, Locality, Content Evolution, and Rates of Change. IEEE/ACM Transactions on Networking, Vol, 12, No. 5, October 2004

[19]. L. Cherkasova, G Ciardo. Characterizing Temporal Locality and its Impact on Web Server Performance. Technical Report HPL-2000-82, Hewlett Packard Laboratories, July 2000.

[20]. M. Arlitt and C. Williamson. Web server workload characterization: the search for invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, May 1996.

katchewan for his insightful suggestions, and thank Chao Jin, Xiongzi Ge and Qiang Zou for their help in writing this paper.

This work is sponsored by the National Basic Research Program of China (973 Program) under Grant No. 2004CB318201, the Key Basic Research Project under Grant No.2004CCA07400, Project CNGI-04-5-1D, Program for New Century Excellent Talents in University NCET-04-0693, the China NSF under Grant No.60273074, No.60503059, No. 60303032, No. 60603048, and the US NSF under Grant No. CCF-0621526.

References

[1]. D. Wenk. Is 'Good Enough' Storage Good Enough for Compliance? *Disaster Recovery Journal*. 2004.

[2]. Q. Zhu and Y. Zhou. Chameleon: A Self-Adaptive Energy-Efficient Performance-Aware RAID. In *Proceedings of the fourth annual Austin Conference on Energy-Efficient Design(ACEED 2005)*, Poster Session, Austin, Texas, March 1-3, 2005.

[3]. D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the Conference on Management of Data*, 1988.

[4]. G. Gibson, Redundant Disk Arrays: Reliable, Parallel Secondary Storage. MIT Press, 1992.

[5]. Q. Xin, E. L. Miller, T. J. Schwarz, D. D. E. Long, S. A.Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, April 2003.

[6]. M. Holland, G. Gibson, and D. Siewiorek. Fast, On-Line Failure Recovery in Redundant Disk Arrays. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 422-43, 1993.

[7]. Jack Y.B. Lee and John C.S. Lui. Automatic Recovery from Disk Failure in Continuous-Media Servers. IEEE Transaction On Parallel And Distributed Systems, Vol. 13, No. 5, May 2002.

[8]. M. Holland, G.A. Gibson and D. Siewiorek. Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays. Journal of Distributed and Parallel Databases, Vol. 2, No. 3, pages 295-335, July 1994.

[9]. R. Hou, J. Menon, and Y. Patt. Balancing I/O Response Time and Disk Rebuild Time in a RAID5 Disk Array. In *Proceedings of the Hawaii International Conference on Systems Sciences*, pages 70-79, 1993.

[10]. Q. Xin, E. L. Miller and T. J. Schwarz. Evaluation of Distributed Recovery in Large-Scale Storage Systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 172-181, June 2004.

[11]. M. Holland. On-Line Data Reconstruction in Redundant Disk Arrays. Carnegie Mellon Ph.D. Dissertation CMU-CS-94-164, April 1994.

[12]. R. Muntz and J. Lui, Performance Analysis of Disk Arrays Under Failure. In *Proceedings of the 16th Conference on Very Large Data Bases*, 1990.

[13]. H. H. Kari, H. K. Saikkonen, N. Park and F. Lombardi. Analysis of repair algorithms for mirrored-disk systems. IEEE Transactions on Reliability, Vol 46, No. 2, pages 193-200, 1997.

[14]. H.M Vin, P.J. Shenoy and S. Rao. Efficient Failure Recovery in Multi-Disk Multimedia Servers. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 12–21, 1995.

[15]. M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, March 2003.

[16]. A. Smith. Cache Memories. Computing Surveys, Vol 14, No. 3, pages 473–480, 1982

[17]. M. E. Gomez and V. Santonja. Characterizing Temporal Locality in I/O Workload. In *Proceedings of the 2002 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'02)*. San Diego, USA, July 2002

[18]. L. Cherkasova and M. Gupta. Analysis of Enterprise Media Server Workloads: Access Patterns, Locality, Content Evolution, and Rates of Change. IEEE/ACM Transactions on Networking, Vol, 12, No. 5, October 2004

[19]. L. Cherkasova, G Ciardo. Characterizing Temporal Locality and its Impact on Web Server Performance. Technical Report HPL-2000-82, Hewlett Packard Laboratories, July 2000.

[20]. M. Arlitt and C. Williamson. Web server workload characterization: the search for invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, May 1996.

[21]. D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of 2000 USENIX Annual Technical Conference*. San Diego, California, USA, June 2000.

[22]. P. G. Sikalinda, L. Walters and P. S. Kritzinger. A Storage System Workload Analyzer. Technical Report CS06-02-00, University of Cape Town, 2006.

[23]. The NetBSD Foundation. The NetBSD Guide. http://www.netbsd.org/guide/en/chap-rf.html.

[24]. The OpenBSD Foundation. The OpenBSD FAQ. http://www.se.openbsd.org/faq/faq11.html#raid.

[25]. The FreeBSD Foundation. The FreeBSD/i386 5.3-RELEASE Release Notes. http://www.freebsd.org/releases/5.3R/relnotes-i386.html.

[26]. W.V. Courtright II, G.A. Gibson, M. Holland and J. Zelenka. RAIDframe: Rapid Prototyping for Disk Arrays. In *Proceedings of the 1996 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '96)*, Vol. 24 No. 1, pages 268-269, May 1996.

[27]. The randread pkg. http://pkgsrc.se/benchmarks/randread

[28]. SPC Web Search Engine I/O Trace. http://traces.cs.umass.edu/storage/

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *;login:,* the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see http://www.usenix.org/membership/specialdisc.html

For more information about membership, conferences, or publications, see http://www.usenix.org.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at http://www.sage.org.

Thanks to USENIX & SAGE Supporting Members

Addison-Wesley Professional/ Prentice Hall Professional
Ajava Systems, Inc.
AMD
Cambridge Computer Services, Inc.
cPacket Networks
DigiCert® SSL Certification
EAGLE Software, Inc.
Electronic Frontier Foundation
FOTO SEARCH Stock Footage and Stock Photography
Google
GroundWork Open Source Solutions
Hewlett-Packard
IBM
Infosys
Intel
Interhack
Microsoft Research
MSB Associates
NetApp
NTT DoCoMo
Oracle
OSDL
Raytheon
Ripe NCC
Sendmail, Inc.
Splunk
Sun Microsystems, Inc.
Taos
Tellme Networks
UUNET Technologies, Inc.
VMware

ISBN-13: 978-1931971508
ISBN-10: 1931971501

90000

9 781931 971508